NEAMS Workbench 1.0 Beta

Bradley T. Rearden
Robert A. Lefebvre
Brandon R. Langley
Adam B. Thompson
Jordan P. Lefebvre

January 2018

Reactor Nuclear Systems Division

NEAMS WORKBENCH 1.0 BETA

Bradley T. Rearden
Robert A. Lefebvre
Brandon R. Langley
Adam B. Thompson
Jordan P. Lefebvre

Date Published: January 2018

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

The mission of the US Department of Energy (DOE) Nuclear Energy Advanced Modeling and Simulation (NEAMS) Program is to develop, apply, deploy, and support state-of-the-art predictive modeling and simulation tools for the design and analysis of current and future nuclear energy systems. This is accomplished using computing architectures that range from laptops to leadership-class facilities. The NEAMS Workbench is a new initiative that will facilitate the transition from conventional tools to high-fidelity tools by providing a common user interface for model creation, review, execution, output review, and visualization for integrated codes. The Workbench can use common user input, including engineering scale specifications that are expanded into application-specific input requirements through the use of customizable templates. The templating process enables multi-fidelity analysis of a system from a common set of input data. Additionally, the common user input processor can provide an enhanced alternative application input that provides additional conveniences over the native input, especially for legacy codes. Expansion of the integrated codes and application templates available in the Workbench will broaden the NEAMS user community and will facilitate system analysis and design. Current and planned capabilities of the NEAMS Workbench are detailed here.

1. INTRODUCTION

The mission of the US Department of Energy (DOE) Nuclear Energy Advanced Modeling and Simulation (NEAMS) Program is to develop, apply, deploy, and support state-of-the-art predictive modeling and simulation tools for the design and analysis of current and future nuclear energy systems. This is accomplished by using computing architectures ranging from laptops to leadership-class facilities. The tools in the NEAMS ToolKit will enable transformative scientific discovery and insights otherwise not attainable or affordable and will accelerate the solutions to existing problems as well as the deployment of new designs for current and advanced reactors. These tools will be applied to solve problems identified as significant by industry, and consequently, they will expand validation, application, and long-term utility of these advanced tools.

The NEAMS program is organized into three product lines: Fuels Product Line (FPL), Reactors Product Line (RPL) and Integration Product Line (IPL).

NEAMS FPL and RPL provide many advanced tools, such as the (1) BISON and MARMOT fuel performance tools based on the Multiphysics Object-Oriented Simulation Environment (MOOSE) from Idaho National Laboratory [1], [2],and (2) the PROTEUS neutronics code and the NEK5000 computational fluid dynamics code from the Simulation-based High-efficiency Advanced Reactor Prototyping (SHARP) tools from Argonne National Laboratory (ANL) [3]. However, these advanced tools often require large computational resources, can be difficult to install, and require expert knowledge to operate, causing many analysts to continue to use traditional tools instead of exploring high-fidelity simulations.

The NEAMS IPL is responding to the needs of design and analysis communities by integrating robust multiphysics capabilities and current production tools in an easy-to-use common analysis environment. This enables end users to apply high-fidelity simulations to inform lower order models used for advanced nuclear system design, analysis, and licensing.

The NEAMS Workbench is a new initiative that will facilitate the transition from conventional tools to high-fidelity tools by providing a common user interface for model creation, review, execution, output review, and visualization for integrated codes [4]. The Workbench can provide a common user input, including engineering-scale specifications that are expanded into code-specific input requirements through the use of customizable templates. The templating process enables multi-fidelity analysis of a system from a common set of input data. Expansion of the codes and application templates available in the Workbench will broaden the NEAMS user community and will facilitate system analysis and design. Users of the Workbench will still need to license and install the appropriate computational tools, but the Workbench will provide a more consistent user experience and will ease the transition from one tool to the next.

2. MULTI-FIDELITY PHYSICS

The NEAMS Workbench will enable analysts to select the fidelity of each type of physics for use in the simulation. Analysts can choose from a variety of tools integrated from many projects and code teams. A conceptual design of tools can be integrated for advanced reactor analysis is shown in Fig. 1. Current production tools with advanced components supported by the NEAMS program—such as those from the Argonne Reactor Codes (ARC) [5] and the SCALE Code System [6]—are shown in light gray, MCNP [7] and other production tools are shown in dark gray, tools from the NEAMS ToolKit are shown in maroon, and tools from Consortium for Advanced Simulation of Light Water Reactors (CASL) [8] are shown in black. The Workbench will make it possible to use the same fundamental engineering input data to create code-specific input models for each tool, enabling analysts to learn more about specific phenomena by performing reference high-fidelity analyses. This will confirm approximations or assumptions in fast-running lower order design calculations.

Multiphysics capabilities provided by tools such as SHARP and MOOSE will continue to provide fully coupled capabilities under the Workbench. The Workbench also provides for traditional single physics codes that form a multiphysics suite with coupling through manipulation of the output of one

code to serve as input for the next. Translation tools will be developed and integrated into the workflow manager to extract and format the needed data in a manner that does not require intrusive changes to the physics codes themselves.
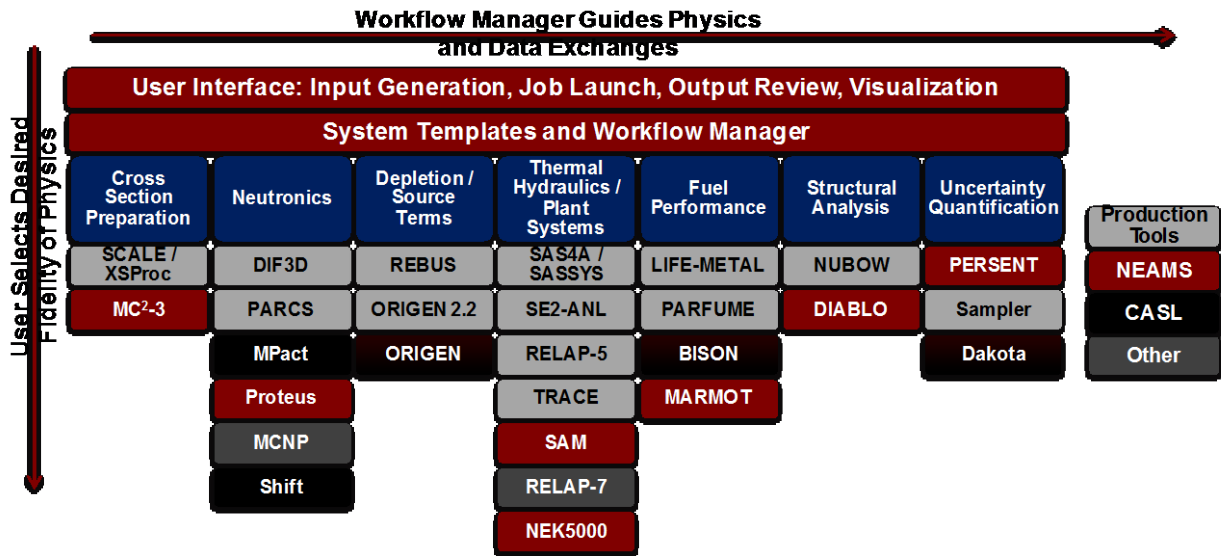
Fig. 1. Conceptual design of tools integrated in the NEAMS Workbench for advanced reactor analysis.

3. USER INTERFACE

To enable users to easily transition from one tool to another, an intuitive user interface is desired. Such an interface will provide guidance on the proper use of a tool by new or infrequent users without impeding the rapid generation or modification of inputs by experiences users. Visualization of input geometry and data files is desired, and the user interface should also provide easy access to computed results in text, tabular, and graphical forms. All integrated codes should be presented with a similar look and feel, and the comparison of results from multiple analyses should be available.

The rapid development and deployment of the NEAMS Workbench was enabled by leveraging the Fulcrum user interface, which was developed over many years and was included in the 2016 release of SCALE 6.2. Fulcrum introduced several new concepts as an integrated user interface for nuclear analysis. It builds on decades of experience from thousands of users, and it replaces eight user interfaces from the 2011 release of SCALE 6.1. Fulcrum is a cross-platform graphical user interface designed to create, edit, validate, and visualize input, output, and data files. Fulcrum directly connects the user with the text form of the input file while providing inline features to assist with building the correct inputs. Fulcrum provides input editing and navigation, interactive geometry visualization, job execution, overlay of mesh results within a geometry view, and plotting of data from file formats. An error checker interactively identifies input errors such as data entry omissions or duplications for all supported codes. The input validation engine identifies allowed data ranges and interdependencies in the input and then reports inconsistencies to the user. The layout of panels in Fulcrum is highly configurable to accommodate the preferences of users, as shown in Fig. 2.
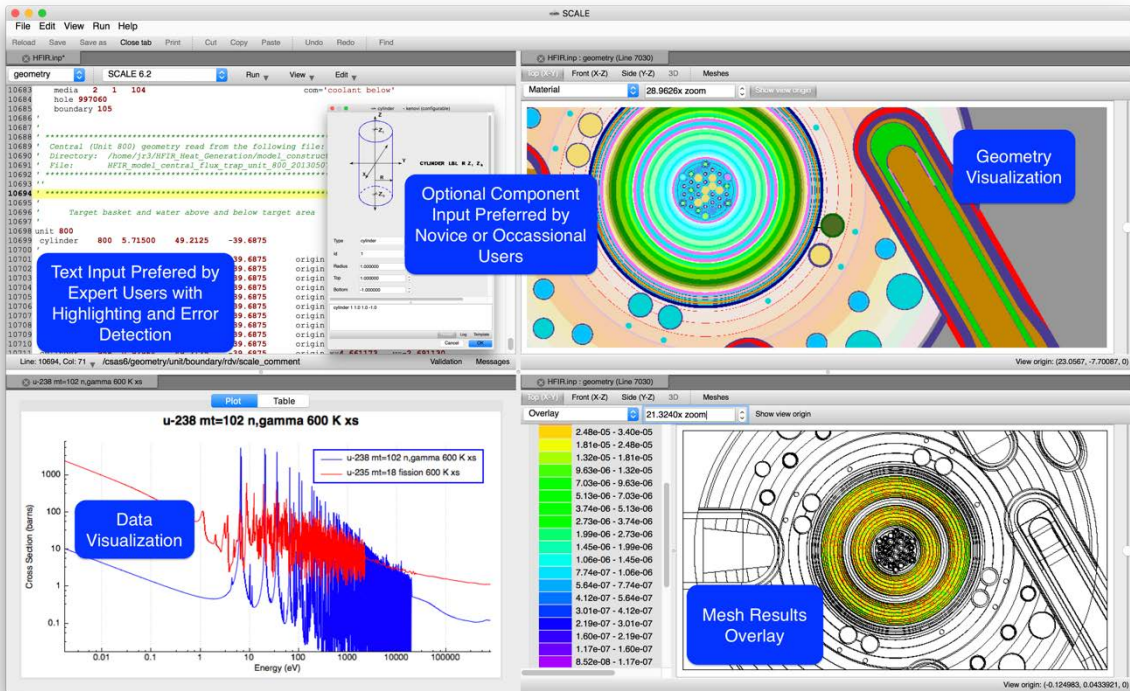
Fig. 2. The NEAMS Workbench leverages the Fulcrum user interface from SCALE.

The configuration files that define how each code input is structured, the requirements and interdependencies of each input block, and the types and content of the dialog boxes available for input assistance are all provided in text files that can be created and edited by any Workbench user. In this way, the members of any team can integrate their tools into the Workbench, whether they are production tools that are widely available, custom-developed analysis tools in a proprietary environment, or university-led prototypes for the exploration of new algorithms. Custom-developed configuration files can be contributed back to Workbench developers to be considered for inclusion in the production version. An example of the interactive input error detection is shown for a SCALE input in Fig. 3, where a user inadvertently typed u02 (u-zero-2) instead of uo2 to specify uranium dioxide as a material. The first error at the bottom of the screen demonstrates how Fulcrum compares the current input against the list of allowed values in this context and recommended alternatives to assist the user. A second error is shown where the user was specifying the uranium enrichment for this material. Here an input rule is applied stipulating that the weight percent values must sum to 100%. In this case, the user inadvertently entered a total of 101%, and Fulcrum identified that error, as well.

Leveraging the configuration files for each code supported by the Workbench, the Fulcrum user interface provides context-aware assistance in auto-completing the input, which can be useful when learning the intricacies of a new code or when transitioning between many codes, as is the intent of the Workbench. An example from SCALE is shown in Fig. 4. In the left image, the user inserts the cursor at the desired point in the input, presses the key combination of *ctrl-space*, and Fulcrum lists all available options in this context, providing geometry shapes in this example. Selecting a *configurable* option launches an additional interactive dialog, shown on the right, to provide additional assistance in populating this portion of the input.

The Fulcrum interface currently supports visualization capabilities from SCALE for Monte Carlo geometry, with results overlaid as shown in Fig. 5. Plotting capabilities are available for nuclear data and covariance data plotting, as well as a variety of computed quantities using line plots, histograms, bar charts, and surface plots, as shown in Fig. 6. The capabilities of Fulcrum have been expanded beyond those required by SCALE, especially to support analysis with NEAMS finite element codes, by integrating visualization capabilities from VisIt [9]. The integration of the ParaView [10] visualization tool is also planned.

3
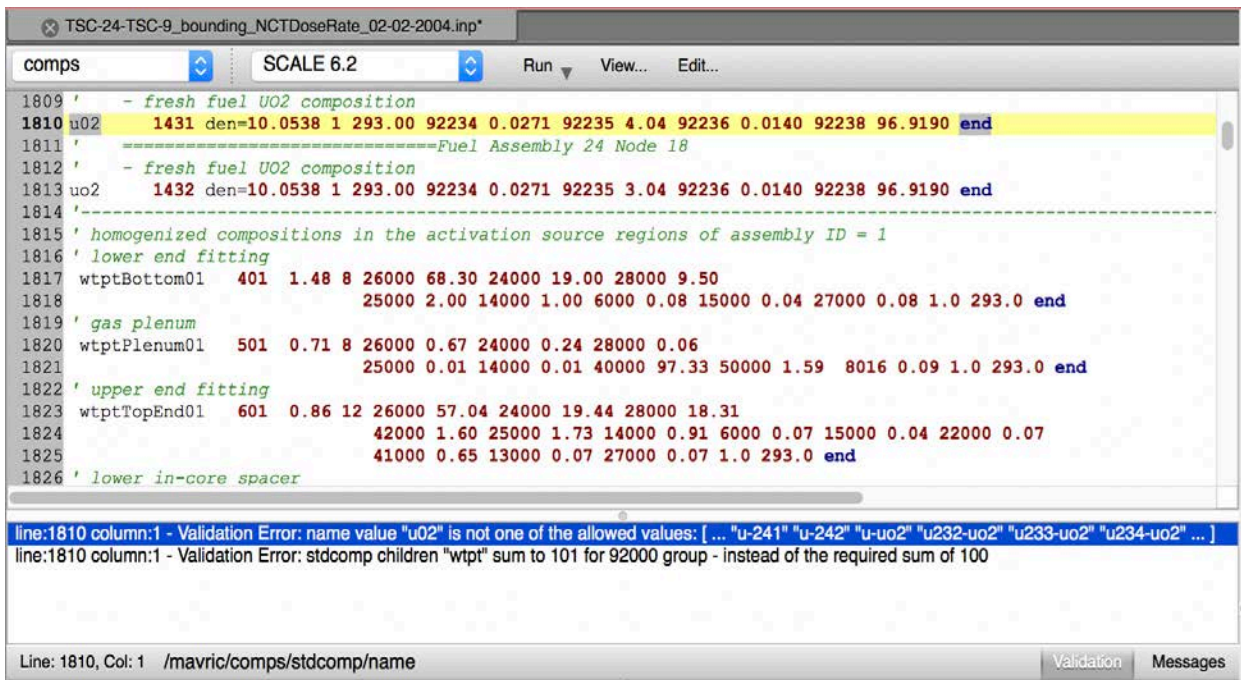
Fig. 3. Fulcrum user interface with interactive input error detection.
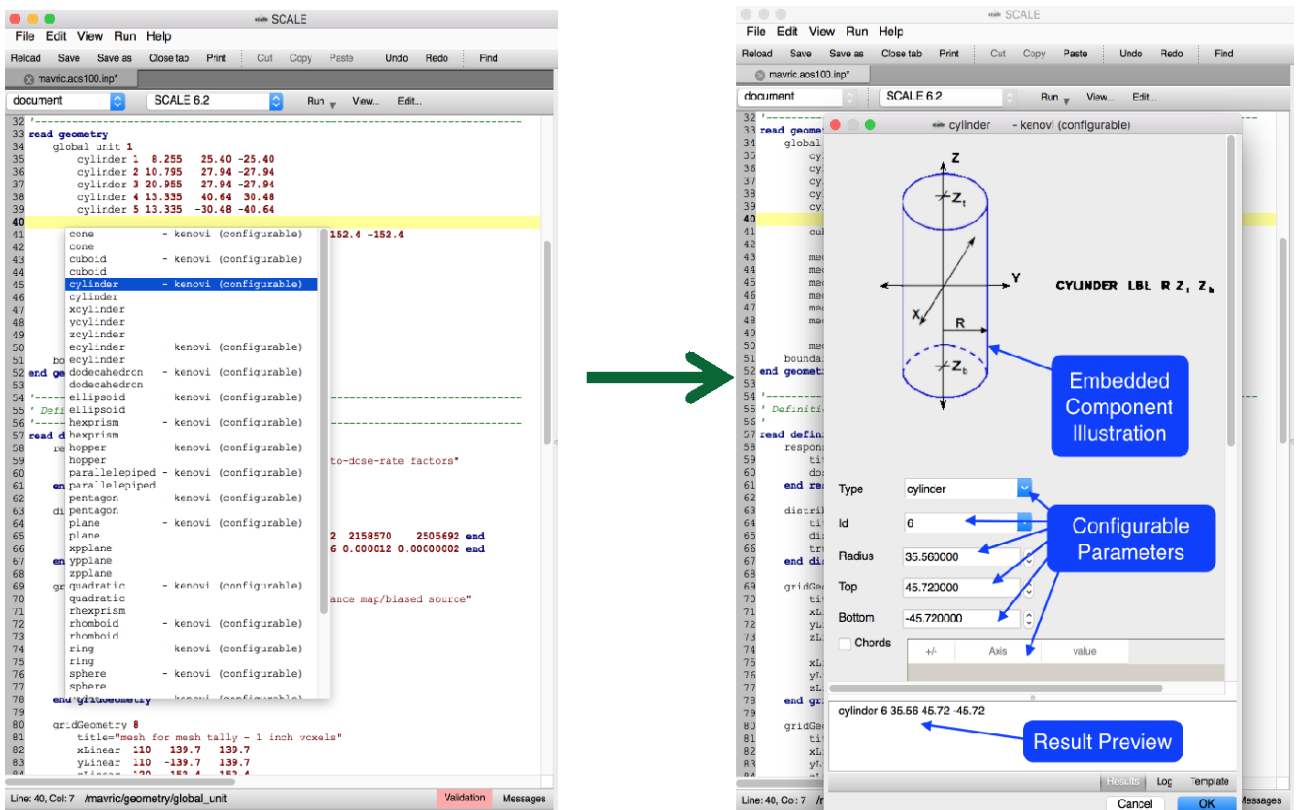
Fig. 4. Fulcrum auto-completion and forms-based input.
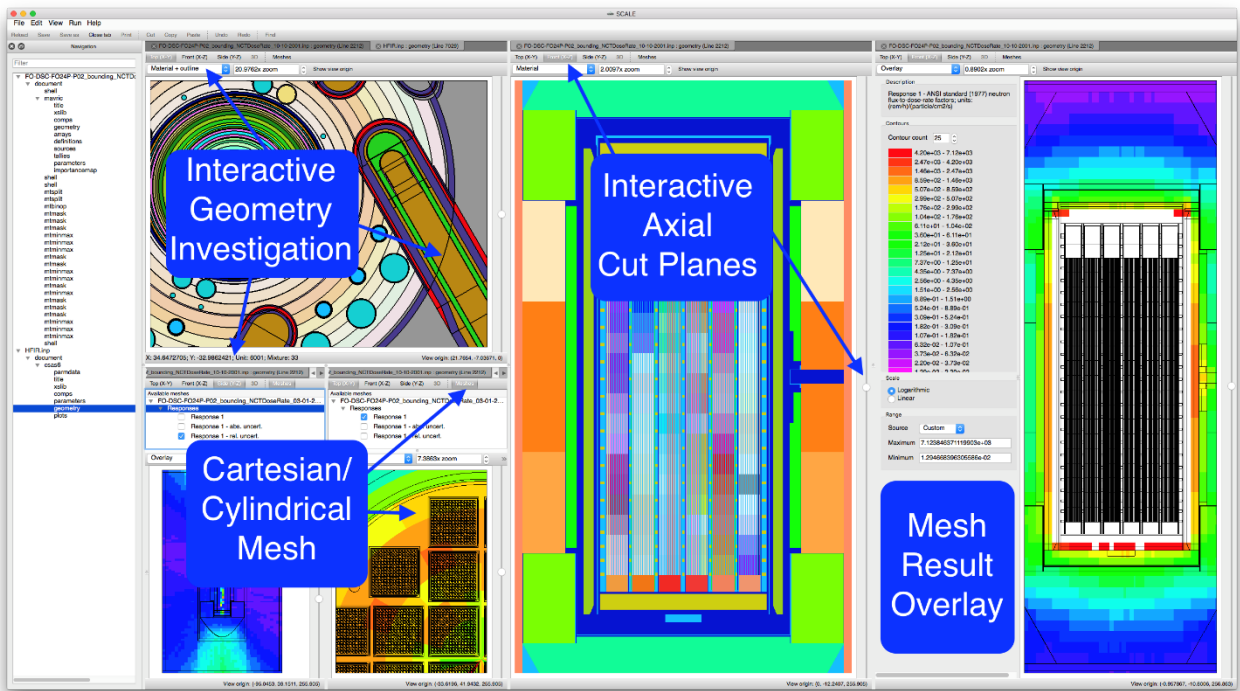
Fig. 5. Fulcrum geometry visualization capabilities.
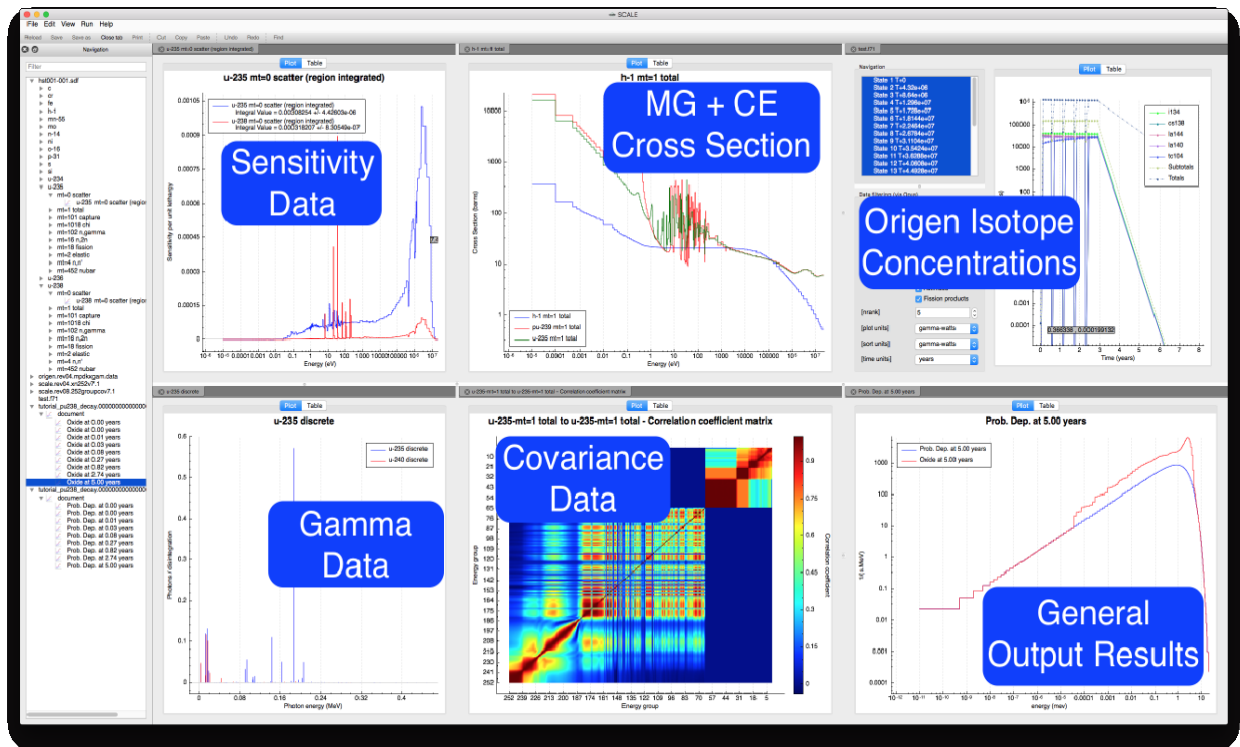
Fig. 6. Fulcrum plotting capabilities.

4. INPUT CONFIRMATION AND ANALYSIS TEMPLATES

The use of multiple tools within the NEAMS Workbench is facilitated through the automatic confirmation of the input format, allowed values, and completeness, as well as through templates that provide for a common definition of a system that can be expanded and run with multiple tools.

The Workbench Analysis Sequence Processor (WASP) [11] package is an open source C++ library and toolset for streamlining the lexical and semantic analysis of ASCII formatted inputs. WASP includes lexers, parsers, and interpreters to provide input processing of integrated applications. WASP uses a parse-tree data structure to facilitate input data retrieval and validation. WASP provides the Workbench with the ability to process common input and data formats. WASP also provides input analysis and templating capabilities through the Hierarchical Input Validation Engine (HIVE) and the Hierarchical Input Template Expansion engine (HALITE).

4.1 HIERARCHICAL INPUT VALIDATION ENGINE (HIVE)

With HIVE, a combination of 19 rules is used describe valid input and to interactively provide users with feedback on the validity of their input. The rules are implemented in an input schema for each supported code. Given an input schema and an input parse-tree from WASP, HIVE will conduct simple type, value, and occurrence checks, as well as complex relational checks. As integrated in the Workbench, HIVE provides immediate feedback to the user in the *Validation* panel, as previously shown in Fig. 3. The HIVE schema files are stored inside the Workbench application in text format, so they can be supplemented or customized by expert users if desired.

The 19 rules of HIVE are as follows:

1. MinOccurs: describes the minimum number of times that an element is allowed to appear under its parent context;
2. MaxOccurs: describes the maximum number of times that an element is allowed to appear under its parent context;
3. ValType: describes the allowed value type for the element (Int, Real, String);
4. ValEnums: describes a list of allowed value choices for the element;
5. MinValInc: describes the minimum inclusive value that this element is allowed to have if it is a number (the provided input value must be greater than or equal to this);
6. MaxValInc: describes the maximum inclusive value that this element is allowed to have if it is a number (the provided input value must be less than or equal to this);
7. MinValExc: describes the minimum exclusive value of the element in the input if it is a number (the provided input value must be strictly greater than this);
8. MaxValExc: describes the maximum exclusive value of the element in the input if it is a number (the provided input value must be strictly less than this);
9. ExistsIn: describes a set of lookup paths into relative sections of the input file and possible constant values where the value of the element being validated must exist;
10. NotExistsIn: describes a set of lookup paths into relative sections of the input file where the value of the element being validated must not exist;
11. SumOver: describes what sum the values must add to under a given context;
12. SumOverGroup: describes what sum the values must add to under a given context when grouped by dividing another input element's value by a given value;
13. IncreaseOver: describes that the values under the element must be increasing in the order that they are read;
14. DecreaseOver: describes that the values under the element must be decreasing in the order that they are read;
15. ChildAtMostOne: describes one or more lists of lookup paths into relative sections of the input file (and possible values) where at most one is allowed to exist;
16. ChildExactlyOne: describes one or more lists of lookup paths into relative sections of the input file (and possible values) where at exactly one is allowed to exist;

17. ChildAtLeastOne: describes one or more lists of lookup paths into relative sections of the input file (and possible values) where at least one must exist;
18. ChildCountEqual: describes one or more lists of lookup paths into relative sections of the input file where the number of values must be equal; and
19. ChildUniqueness: describes one or more lists of lookup paths into relative sections of the input file where the values at all of these paths must be unique.

These rules facilitate all input validation tasks with the exception of higher order validation tasks such as comparing input data with data from a related binary file, or validating constructive solid geometry parameters. At this time, the validation engine cannot handle recursion (e.g., validation of mathematical expression).

4.2 HIERARCHICAL INPUT TEMPLATE EXPANSION ENGINE (HALITE)

The Hierarchical Input Template Expansion Engine (HALITE) couples flat or hierarchical data with application-specific input templates to facilitate Workbench input auto-completion and future workflow tasks. The HALITE engine provides Workbench with the capability to create application-specific input using application-agnostic data. HALITE provides common attribute and expression substitution, as well as the unique capability to drive input expansion via a single hierarchical data set. This is the same data-driven workflow construct that has been successfully demonstrated in the UNF Storage, Transportation & Disposal Analysis Resource and Data System (UNF-ST&DARDS) [12].

HALITE has the following templating features:

- Scalar and iterative attribute and mathematical expression substitution enables traditional and complex data insertion;
- Substitution formatting enables rigid fixed-width input formats where needed;
- Conditional template blocks enable inclusion or exclusion of template sections using some condition;
- File input enables reuse of a common sub-template; and
- Parameterized file input enables advanced reuse of a sub-template with explicit loop or implicit array element iterative repetition or specific data sets.

A sketch of the HALITE workflow to expand a common problem definition in terms of engineering parameters such as dimensions, compositions, etc. into code specific inputs is show in Fig. 7.
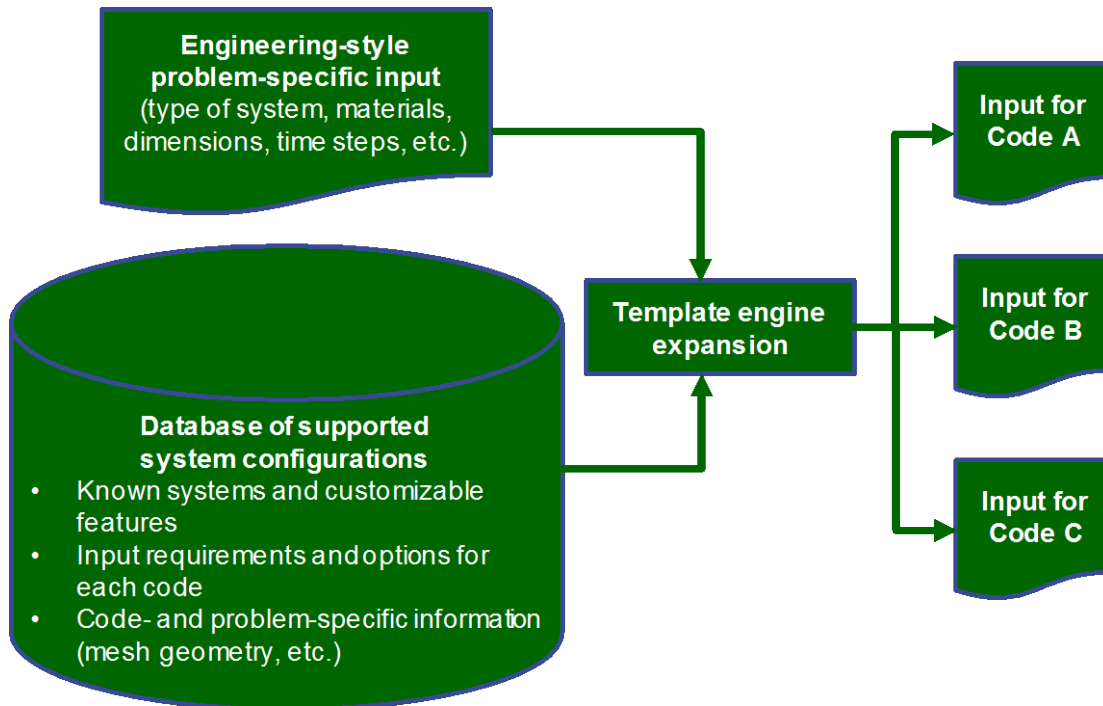
Fig. 7. Sketch of HALITE template expansion to provide input for multiple codes from the same problem definition.

5.   VISUALIZATION TOOLS

The interpretation of input data, geometry models, and output results is enabled through the convenient integration of multiple visualization tools. The VisIt tool is embedded in the Workbench for visualization of 3D mesh geometry and mesh results. Customized plotting is enabled for any application using Fulcrum's native plotting package that has been extended with a customizable interface to any text file.

5.1   VISIT

VisIt is a powerful distributed, parallel visualization and graphical analysis tool developed for analysis of mesh data from high-performance computing. In the NEAMS Workbench, VisIt is composed of three parts:

1. **The VisIt Graphical User Interface (GUI)**, which will load as an embedded application in its own dockable panel. The native VisIt controls are fully accessible from the VisIt GUI
2. **The VisIt Canvas**, which is a tab within the NEAMS Workbench workspace layout
3. **VisIt visualization**, which requires a VisIt Canvas. A new VisIt visualization will automatically create a new VisIt Canvas for visualization and graphical analysis of the mesh data.

The integration of VisIt provides an important extension of the previous Fulcrum visualization capabilities that did not allow for plotting of common 3D mesh formats such as Exodus, VTK, and the broad range of formats and features available in VisIt. Figure 8 depicts VisIt embedded in the NEAMS Workbench with a flexible window layout.
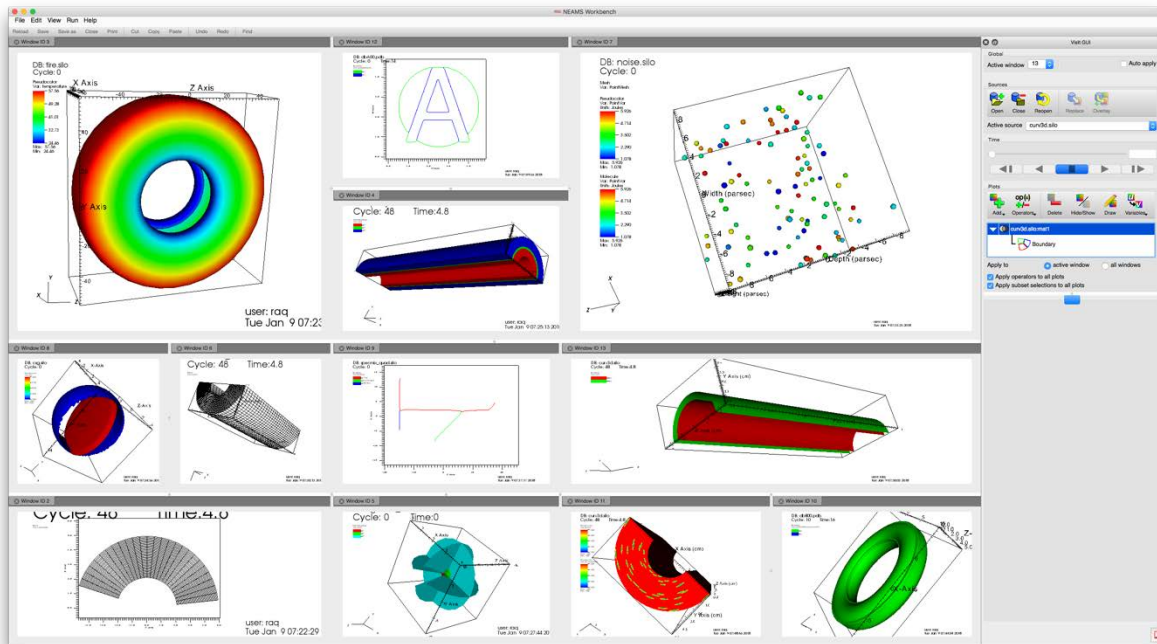
Fig. 8. VisIt visualization embedded in the NEAMS Workbench.

5.2   EXTENDABLE PLOTTING

Because the NEAMS Workbench is intended to support a broad range of codes, any of which may have its own binary or ASCII data formats, an extendable data processor plotting capability was developed to enable the Workbench to extract data from any text file and create plots such as those shown in Fig. 6. Any developer or user can create a Workbench processor file that will extract data from a text file to create a NEAMS Workbench data plot. The processor file is interpreted at runtime, allowing the user to extend the NEAMS Workbench plotting capabilities for local installation. The processor file can be shared with colleagues, further enabling collaboration.

The Workbench processor files are composed of three components: (1) processor hierarchy, (2) processor filtering, and (3) the processor engine. Because there could be numerous processor files associated with different types of analysis, an optional hierarchy feature is available to allow the user to dictate the organization of the processors in the Workbench user interface. This hierarchy allows many processors to be binned into fewer logical groups. In addition to processor organization via hierarchy, the extensions and filter_pattern features allow the user to limit the files for which the processor is enabled according to the given file extensions and/or the specified filter_pattern. The ability to limit when the processor is enabled ensures that the NEAMS Workbench is streamlined for the data under inspection. The processor consists of plot series information and data extraction logic. The data extraction logic uses any command line utility to extract data into spreadsheet format. The *Grep* and *Awk* utilities are most commonly used and are available across all supported platforms. The data extraction logic can create multiple series, one per spreadsheet. The plot series can reference the data using a familiar, Excel-like cell-reference mechanism which allows for capturing the keys, values, and uncertainties (low, high). Line style, axis labels, and scale can all be specified. In addition to 2D line and scatter graphs, bar charts and color map plotting are also available. Figure 9 depicts application output plotting enabled by integrated Workbench processors.
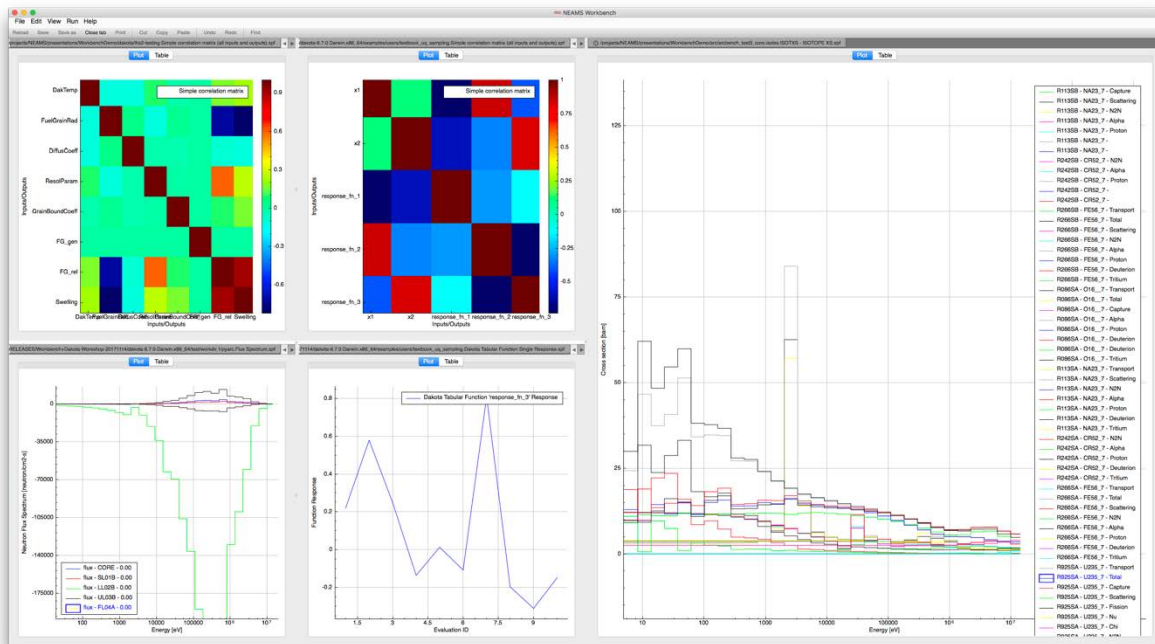
Fig. 9. Extendable Workbench processor plotting.

6.   RUNTIME ENVIRONMENT

A goal of the NEAMS Workbench is to facilitate transition from conventional tools to high-fidelity tools. Many codes involve different means of launching a calculation, and some have multiple means. A generic runtime environment interface was created to facilitate their execution and to provide a consistent interface through which the user can interact with each tool in all necessary modes of operation (e.g., serial, parallel, and scheduled execution).

Tools with no runtime environments require the user to manually conduct all steps associated with running them. For example, the user might be required to copy the problem input file into a specific location with a specific file name, such as *input*, and then invoke the application, thus producing temporary scratch files and output file(s). If multiple simultaneous calculations are desired, many complications follow that will likely lead to failed or erroneous results. Other codes are distributed with sophisticated job management capabilities designed for use in quality assured licensing calculations that should not be disrupted. As such, the NEAMS Workbench enables the integration of customized runtime environments for each integrated tool. A runtime script provides the setup, execution, or finalization logic needed to fulfill the runtime interface. The setup logic might create a working directory, *TMPDIR*, and then copy the *problem.inp* into the *TMPDIR* as *TMPDIR/input*. The execution might invoke the application executable, passing application messages back to the calling application (e.g., command console, Workbench). The finalized logic might (1) combine the output files located in *TMPDIR* into logical order, (2) copy the output back into *problem.out*, residing next to *problem.inp*, and (3) delete the *TMPDIR* to clean up after itself. Once integrated by a developer or expert users, runtime scripts are accessible through a dropdown menu in Workbench. The scripts are written in Python and stored in the runtime environment (*rte*) directory inside of Workbench, so they can be customized and supplemented as tools and use scenarios evolve.

A Workbench script can be developed to allow consistent invocation for any specific application logic, and it may provide great convenience relative to the application's typical command-line interface. As the runtime environment matures and additional features (e.g., a queuing system) are added for the base class, all incorporated runtimes will benefit.

7. CODE INTEGRATION

To facilitate the use of NEAMS-developed codes, as well as many commonly used production capabilities, a wide range of tools from many teams was integrated for this first beta release of the NEAMS Workbench. Integration of a new code is enabled by first ensuring that WASP can process its input format. For modern tools that use common input structures, this first step is fairly simple. For legacy codes that use custom developed input formats, new customized features in WASP must be developed. Once the input can be read into Workbench in a hierarchical format, the HIVE schema files are developed, generally by following the user documentation to support various blocks of input, confine input to allowed values, and manage interdependencies between multiple input parameters. Next, a runtime environment script is developed that includes processes to properly arrange the input files and executables and to retrieve the output data. HALITE templates are developed and added to the Workbench configuration to provide autocomplete of all or part of application's input. At any time, a software developer or end user can customize or supplement the HIVE schema files or HALITE templates for codes with input formats supported by WASP. The codes integrated for the NEAMS Workbench Beta 1.0 release are detailed below.

7.1 MOOSE APPLICATIONS

The MOOSE framework provides the foundation for many NEAMS developed tools, most visibly the BISON fuels performance code. Initial efforts for the NEAMS Workbench focused on convenient coupling between the MOOSE framework and the NEAMS Workbench. With its modern software design, MOOSE includes a common input format and a common engine for processing input for any MOOSE application. WASP was updated to support the MOOSE input format as one of its known input styles, and MOOSE was updated to generate a HIVE schema file for any MOOSE application. Simply running a MOOSE application with the command line argument --definitions will generate a HIVE schema file representing the current input features of the MOOSE application. This file is then added to the Workbench configuration prior to startup. A generic MOOSE runtime is provided with the NEAMS Workbench. A custom runtime environment must only be developed for each MOOSE application if the command line arguments are not already available in the MOOSE Workbench generic runtime. As BISON is a key NEAMS tool, it is integrated in the NEAMS Workbench Beta 1.0 with a schema file and runtime script, and other MOOSE applications will be integrated in future releases or can be added by end users. After running a MOOSE calculation, the output file can be viewed, and the mesh data files can be visualized with the embedded VisIt tools, as shown in Fig. 10 for a BISON calculation.
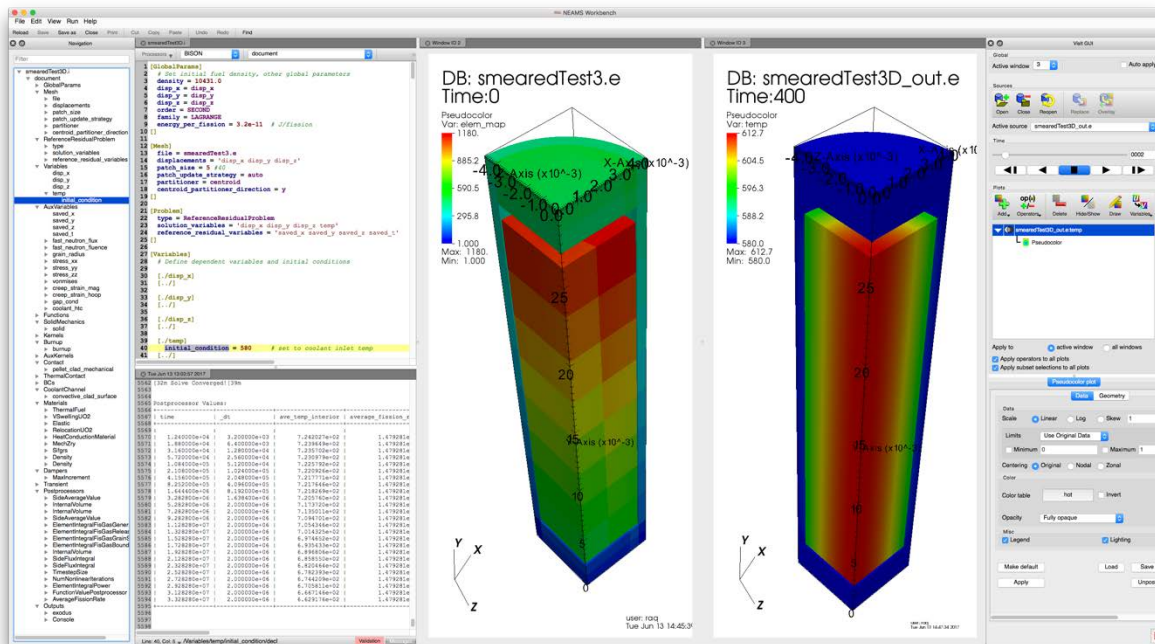
Fig. 10. BISON integration in the NEAMS Workbench.

7.2 DAKOTA

The Dakota suite of iterative mathematical and statistical methods has been integrated into the Workbench. The suite is from Sandia National Laboratories, and the methods interface with many computational models. A new definition-driven input interpreter was developed and added to WASP to support the Dakota input format, and many updates were implemented in the internal Dakota schema file to provide for improved consistency in the Dakota input. A Python translation script was written to convert the internal Dakota input schema to a HIVE schema file. A customized runtime environment was created, and customized plotting capabilities were enabled to allow for convenient visualization by extracting data from the output file and generating plots. An example calculation of Dakota in the NEAMS Workbench is shown in Fig. 11.
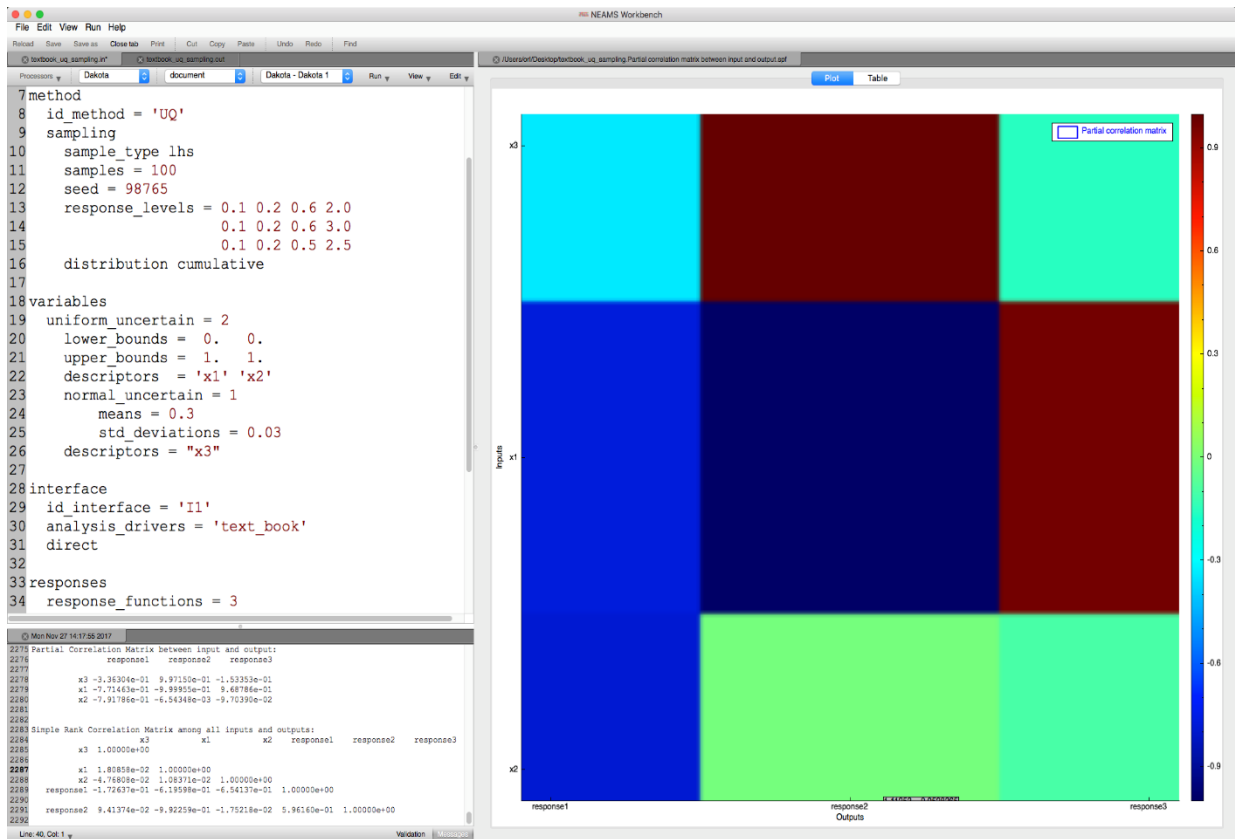
Fig. 11. Dakota integration in the NEAMS Workbench.

7.3 ARGONNE REACTOR CODES

The traditional Argonne Reactor Codes (ARC) for fast reactor analysis have been integrated into the NEAMS Workbench at the request of the Advanced Reactor Technologies team members who routinely use these tools. In this work, a new common input was developed using the Standard Object Notation input format developed for SCALE 6.2 and revised in WASP. This common input is used to populate templates to run calculations with MC^2-3, DIF3D, REBUS, and PERSENT for integrated, problem-dependent, cross section preparation, core analysis, depletion, and sensitivity/uncertainty analysis. Before the common input was developed, each of these calculations required a separate input file, and many of the input formats were so difficult to use that a specialized script was often required to generate them. With the ARC/Workbench integration creating the PyARC [13] Workflow manager, users can easily create engineering style input, have the Workbench generate the input for the codes, launch the calculations using a customized runtime environment, and visualize the results with the embedded Workbench processor files and VisIt tool. An integrated ARC/Workbench input and associated workflow are shown in Fig. 12. An example ARC calculation in the NEAMS Workbench is shown in Fig. 13. Users who access the ARC codes through the Workbench will have easier access to advanced codes of NEAMS, which in the future will be able to leverage common input parameters, especially with the planned integration of MCNP and PROTEUS.

13
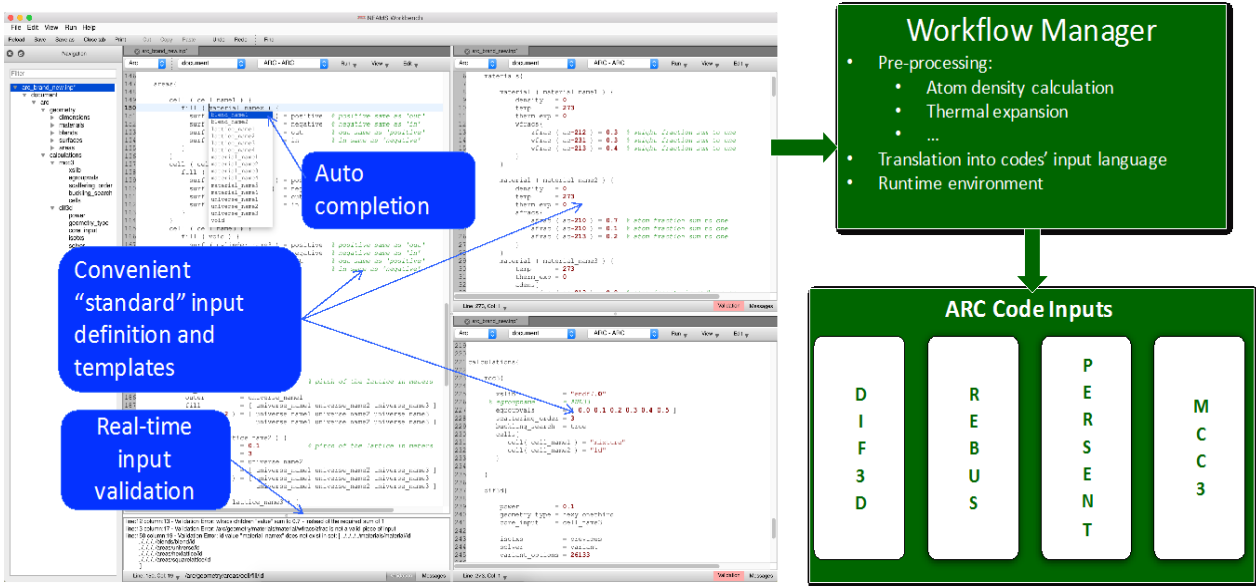
Fig. 12. Prototype ARC/Workbench integration and associated workflow manager.
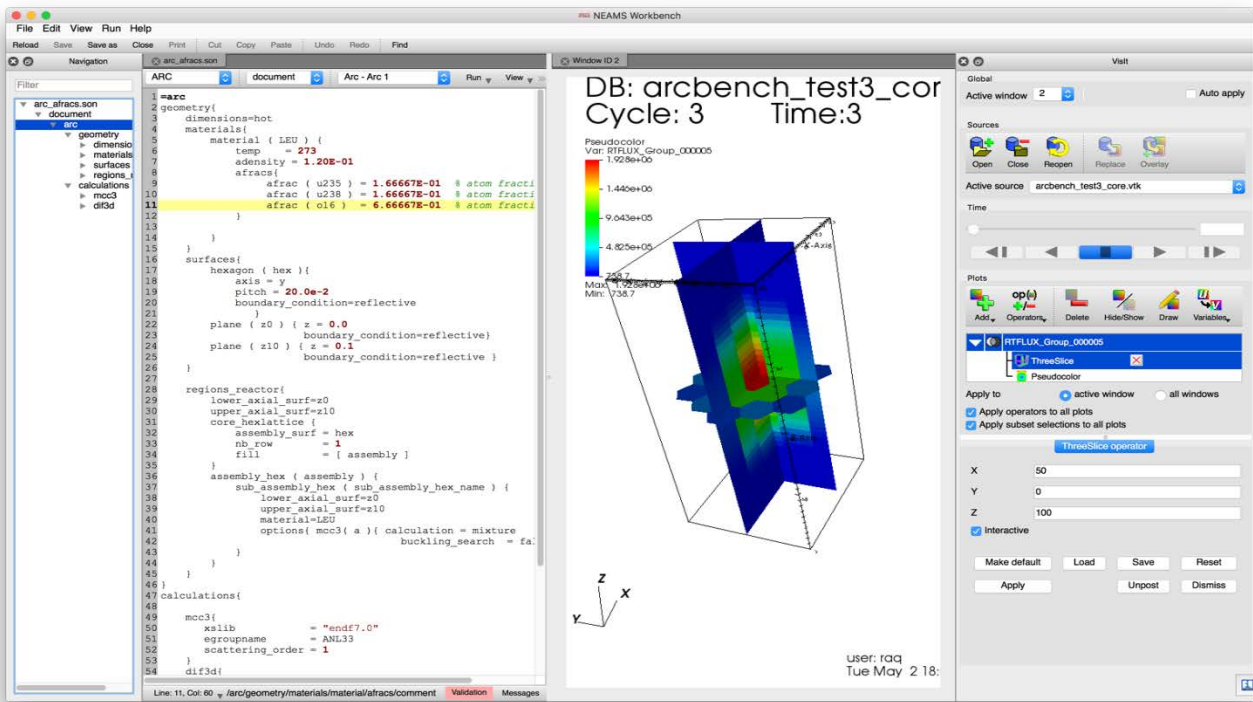
Fig. 13. ARC integration in the NEAMS Workbench.

7.4 SCALE CODE SYSTEM

Because the NEAMS Workbench leverages the Fulcrum user interface from SCALE, it supports the dozens of verified and validated design and licensing tools used for criticality safety, reactor physics, radiation shielding, radioactive source term characterization, and sensitivity/uncertainty analysis for a full range of systems light water reactors (LWRs), advanced reactors, and research/test reactors. Examples of SCALE integration are shown in Fig. 3 – Fig. 6 above.

7.5 CODE INTEGRATION

Table 1 includes the codes that are integrated for the NEAMS Workbench Beta 1.0 release, as well as near-term code integration activities. Under a Nuclear Energy University Program award, Rensselaer Polytechnic Institute (RPI) is partnering with ORNL's Workbench team, ANL's PROTEUS team, and the Los Alamos National Laboratory (LANL) MCNP team to integrate these tools. Teams at ORNL are leading efforts to integrate the NEK5000 code, as well as the Warthog tool that couples PROTEUS neutronics with BISON fuel performance. Additionally, the Systems Analysis Module (SAM) from ANL will be integrated in the near future.

Table 1. Code Integration for the NEAMS Workbench

Tool	Application	Status	Integration lead
MOOSE	General purpose multiphysics framework	Beta 1.0	ORNL/INL
BISON	Fuel performance	Beta 1.0	ORNL/INL
Dakota	Uncertainty quantification and model optimization	Beta 1.0	SNL/ORNL
ARC	Fast reactor analysis	Beta 1.0	ANL
SCALE	Widely used multipurpose neutronics and shielding analysis	Beta 1.0	ORNL
PROTEUS	Three-dimensional unstructured grid finite element neutron transport solver	In progress	RPI/ANL
MCNP	Widely used Monte Carlo radiation transport code	In progress	RPI/LANL
NEK5000	Computational fluid dynamics analysis	In progress	ORNL
Warthog	Multiphysics neutronics and fuel performance in MOOSE	In progress	ORNL
SAM	MOOSE tool for single phase systems analysis	Planned	ORNL/ANL
VERA-IN	CASL multiphysics tools	Planned	ORNL

8. AVAILABILITY

The NEAMS Workbench Beta 1.0 is available to interested users and developers. An open source version is planned for the near future, but because of the origins of the Fulcrum user interface and its integrated development with SCALE, a SCALE license is currently required to request the beta version of the NEAMS Workbench. The user documentation for operating many Workbench features is provided in Appendix A of this report.

Instructions for acquiring the NEAMS Workbench 1.0 Beta release are as follows:

1. Acquire RSICC License for the SCALE Code System
   https://rsicc.ornl.gov/codes/ccc/ccc8/ccc-834.html
2. Email nwb-help@ornl.gov to request NEAMS Workbench
   a. Include your SCALE RSICC license confirmation email
   b. Include the operating system(s) for which you would like prebuilt binary executables

3. Receive a response from nwb-help@ornl.gov
   a. Download the NEAMS Workbench deployment
   b. Follow the installation instructions

9.   CONCLUSIONS

The NEAMS Workbench is a new initiative created in response to the needs of design and analysis communities. It is intended to enable end users to apply high-fidelity simulations to inform lower order models for the design, analysis, and licensing of advanced nuclear systems. In its beta release capacity, it enables enhanced input editing, validation, and navigation capabilities to assist new users of NEAMS tools in getting started. In addition, the NEAMS Workbench provides basic job execution via an extensible runtime environment and convenient output visualization and analysis capability via fast 2D data plotting, as well as 3D mesh visualization through the integrated VisIt toolkit.

The NEAMS Workbench will facilitate the transition from conventional tools to high-fidelity tools by providing a common user interface and common user input processing capabilities: flexible input formats, hierarchical validation engine, and template engine. An extensible runtime environment ensures that computational environments can be taken into account. Integrated applications and associated system templates will continue to broaden the NEAMS tools user community and will facilitate system analysis and design.

The future open source release of the NEAMS Workbench user interface and common input processing capabilities will further enable future collaboration and will enable extensions of NEAMS Workbench for proprietary industry application modeling and simulation needs.

10.   ACKNOWLEDGMENTS

11. REFERENCES

1. D. Gaston, C. Newman, G. Hansen, And D. Lebrun - Grandié, "MOOSE: A parallel computational framework for coupled systems of nonlinear equations, *Nucl. Eng. Des.*, **239**, 1768–1778 (2009).

2. R.L. Williamson, J.D. Hales, S.R. Novascone, M.R. Tonks, D.R. Gaston, C.J. Permann, D. Anders, and R.C. Martineau, "Multidimensional multiphysics simulation of nuclear fuel behavior," *J. Nucl. Mater*, **423** 149–163 (2012).

3. Y.Q. Yu, E.R. Shemon, J.W. Thomas, Vijay S. Mahadevan, Ronald O. Rahaman, and Jerome Solberg, *SHARP User Manual*, ANL-NE-16/6, Argonne National Laboratory, Argonne, Illinois (2016).

4. B.T. Rearden, R.A. Lefebvre, A.B. Thompson, B.R. Langley, and N.E. Stauff, "Introduction to the Nuclear Energy Advanced Modeling and Simulation Workbench," *M&C 2017 – International Conference on Mathematics & Computational Methods Applied to Nuclear Science and Engineering*, Jeju, Korea, April 16–20, 2017.

5. *ARC 11.0: Code System for Analysis of Nuclear Reactors*, Argonne National Laboratory (2014). Available from Available from Radiation Safety Information Computational Center as CCC-824.

7. D.B. Pelowitz, Ed., *MCNP6 User's Manual, Version 1.0,* Los Alamos National Laboratory report LA-CP-13-00634, Rev. 0 (May 2013). Available from Radiation Safety Information Computational Center as CCC-810.

8. John A. Turner, Kevin Clarno, Matt Sieger, Roscoe Bartlett, Benjamin Collins, Roger Pawlowski, Rodney Schmidt, and Randall Summers, "The Virtual Environment for Reactor Applications (VERA): Design and Architecture," *J. of Comput. Phys.,* **326**, 544–568 (2016).

9. LLNL: VisIT Visualization Tool (2002–2016). https://wci.llnl.gov/codes/visit

10. Kitware: ParaView (2002–2016). http://www.paraview.org

11. R.A. Lefebvre, B.R. Langley, and J.P. Lefebvre, *Workbench Analysis Sequence Processor*, ORNL/TM-2017/619, UT-Battelle, LLC, Oak Ridge National Laboratory (2017).

12. R.A. Lefebvre, J.M. Scaglione, J.L. Peterson, P. Miller, G. Radulescu, K. Banerjee, K.R. Robb, A.B. Thompson, and J.P. Lefebvre, "Development of Streamlined Nuclear Safety Analysis Tool for Spent Nuclear Fuel Applications," *Nuclear Technology* Vol. 199, Issue 3, 2017.

13. N. Stauff, N. Gaughan, and T. Kim, "ARC integration into the NEAMS Workbench," ANL/NE-17/31, September 30, 2017.

**APPENDIX A. NEAMS WORKBENCH 1.0 BETA USER
DOCUMENTATION**

APPENDIX A. NEAMS WORKBENCH 1.0 BETA USER DOCUMENTATION

NEAMS Workbench User Documentation

version 1.0 beta
NEAMS Workbench Help Documentation

Robert A. Lefebvre

Adam B. Thompson

Brandon R. Langley

Jordan P. Lefebvre

October 31, 2017

Appendix Table of Contents

NEAMS Workbench User Documentation

Welcome to the NEAMS Workbench user documentation. You are here for two reasons.

1. To use NEAMS Workbench to help you run an application. Options are currently:

   - SCALE - https://rsicc.ornl.gov/codes/ccc/ccc8/ccc-834.html
   - ARC - https://rsicc.ornl.gov/codes/ccc/ccc8/ccc-824.html and the PyARC interface
   - MOOSE Applications - http://mooseframework.org/
   - Dakota - https://dakota.sandia.gov/

2. To integrate an application into NEAMS Workbench

   - Read Application Integration section
   - Click the `help>Documents` training Session*.pdf
   - Contact `NEAMS Workbench Help <nwb-help@ornl.gov>`

There are several steps to get started using NEAMS Workbench with an integrated application.

1. Create an application runtime configuration via *Configurations*
2. Open or create an application input file. This may require the selection of the application's grammar, more of which can be learned from the *Grammar Support* section.
3. Edit, navigate, validate, execute, and review.

We know there is room for improvement, so please help us prioritize future work by providing feedback.

Requirements

Supported Operating Systems

NEAMS Workbench Help <nwb-help@ornl.gov>.

- Linux 64-bit (Fedora, RHEL6)
- Mac OS X (Darwin) 10.9.5 or newer
- Windows 7,10 64-bit

For help compiling NEAMS Workbench from the source, please send an email to NEAMS Workbench Help at nwb-help@ornl.gov.

System Requirements

- Minimum requirements: 2 GB RAM, dual core processor
- Recommended requirements: 4+ GB RAM, quad core processor
- Recommended for large models: 16 GB RAM

Features

Settings

This section outlines the growing number of ways the user is able to customize Workbench's environment and interface.

Note
The settings editor is accessible via File -> Settings...

Environment

Workbench requires the following resources to be able to properly facilitate user actions (component creation, standard composition listing, etc.).
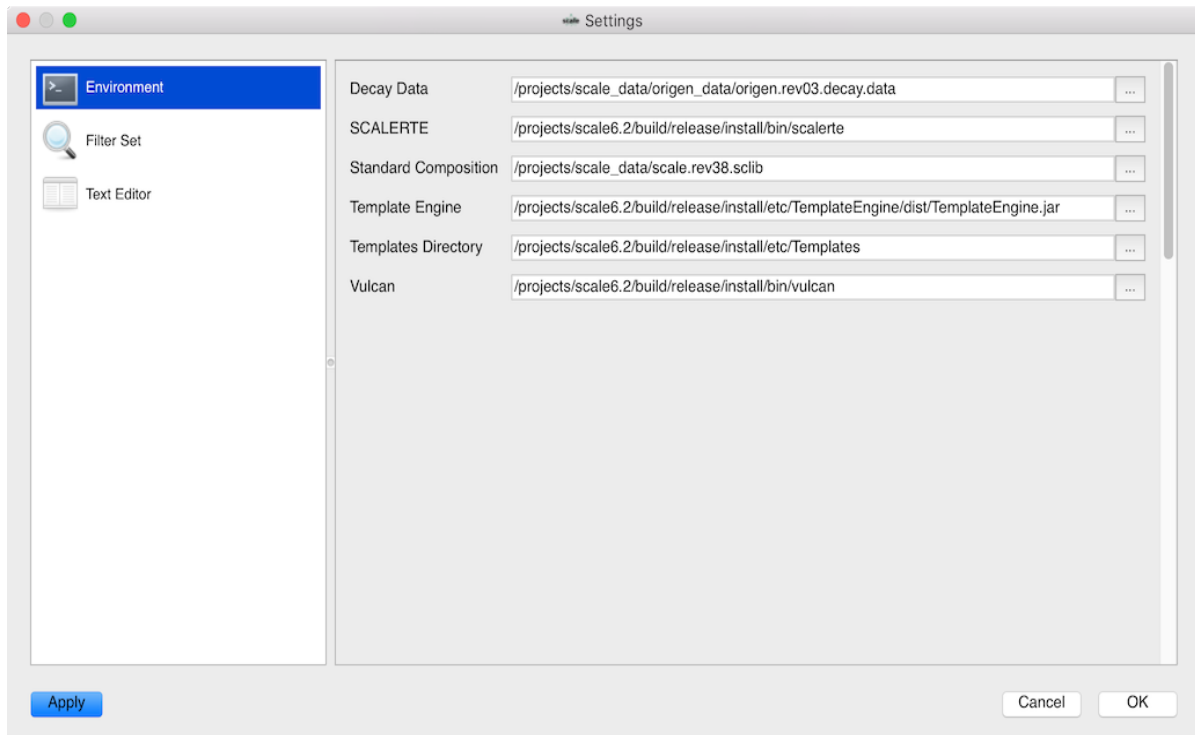
Decay Data This is needed to conduct unit conversions for data. This file is typically located in <path to SCALE>/data/origen.rev*.decay.data

SCALERTE The SCALE runtime environment to which Workbench delegates execution of SCALE inputs

Standard Composition This is the SCALE standard composition library from which Workbench provides composition introspection to facilitate component creation.

Template Engine The Java application which facilitates template expansion. *Template expansion* is an integral part of component creation and the ORIGAMI Automator.

Templates Directory This is the directory in which templates reside. These templates—and potentially their Fulcrum User Interface (FUI) files—are used to facilitate component creation.

Vulcan This is the SCALE material processor. This command line utility is a convenience application to the SCALE material processor. Vulcan converts SCALE composition/mixture/material input into mixture, nuclide, and number density triplets.

Filter Set Workbench provides four preset lists of nuclides with which the user may filter what is being plotting while viewing the contents of an F71 file. These lists may be edited by the user in the settings dialog, but currently, the user can only add or remove a preset by directly editing the settings file on disk.
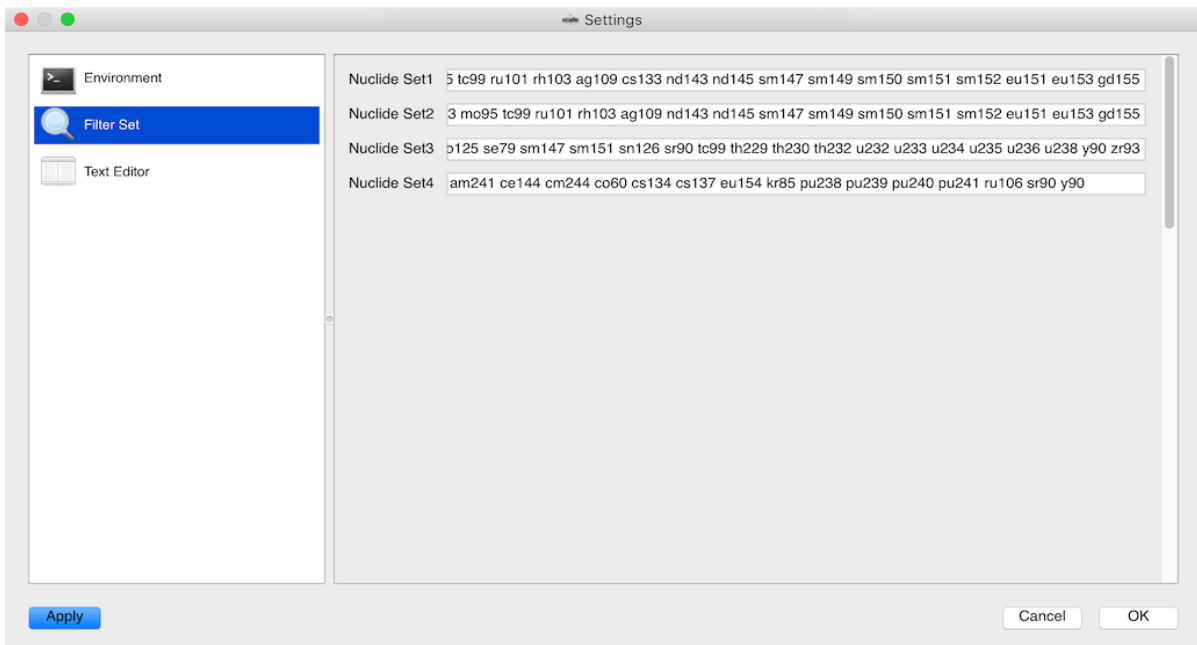
Text Editor

One of Workbench's core capabilities is editing any text file the user would like to edit. This group of settings allows the user to customize the text editing experience, including specifying the font, syntax highlighting colors, etc. These settings apply to all editors.
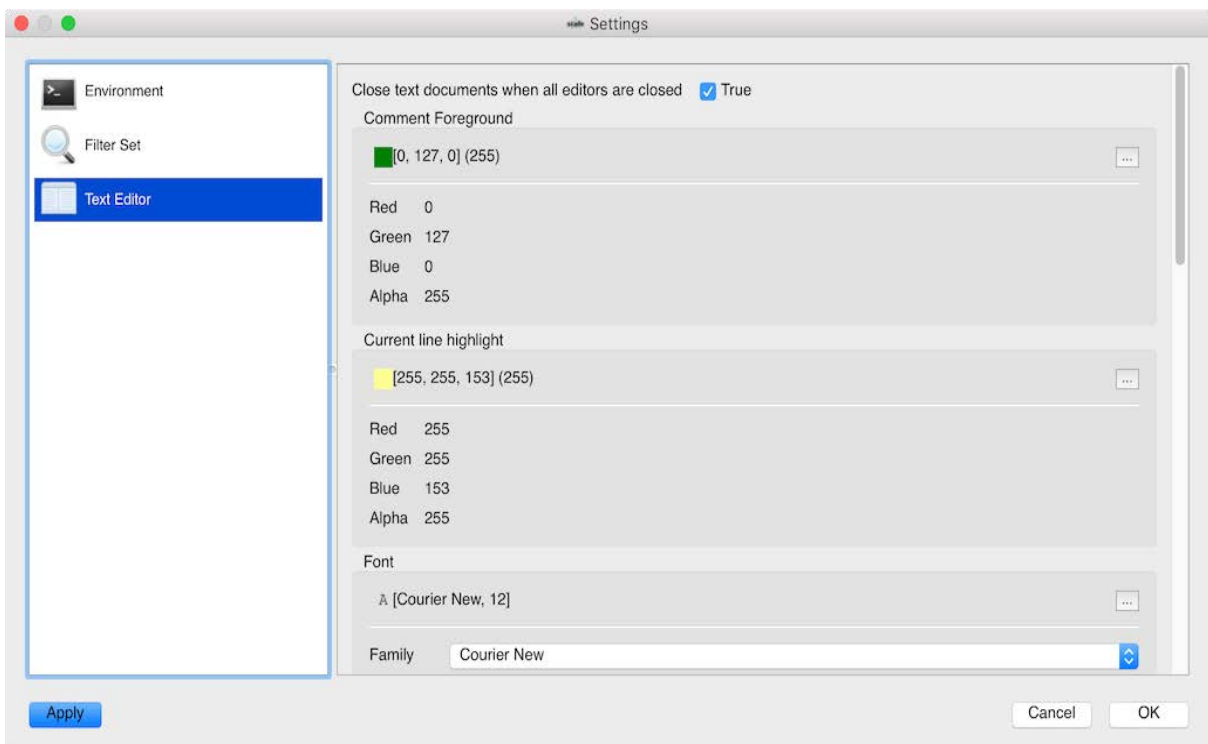
Close text documents when all editors are closed This indicates whether to close the document if all associated text editors have been closed. "False" allows the document to stay open, displayed in the navigation tree, until the user explicitly closes the document (via the File menu or other documented manners).

Comment Foreground

The foreground color of any text identified as a comment.

```
30 '-------------------------------------------------------------------------
31 ' Geometry Block - SCALE standard geometry package (SGGP)
32 '-------------------------------------------------------------------------
```

Current Line Highlight

This is the background color of the line on which the cursor currently resides.

```
33 read geometry
34     global unit 1
35         cylinder 1  8.255    25.40 -25.40
```

Font

The font specifies the text editor's current font.

Highlight Current Line Indicates whether the line on which the cursor currently resides should be highlighted with the color specified by Current Line Highlight.

Keyword Foreground The foreground color of any text identified as a keyword.

```
33 read geometry
```

Number Foreground The foreground color of any text identified as a number.

```
34     global unit 1
35         cylinder 1  8.255    25.40 -25.40
36         cylinder 2 10.795    27.94 -27.94
37         cylinder 3 20.955    27.94 -27.94
38         cylinder 4 13.335    40.64  30.48
39         cylinder 5 13.335   -30.48 -40.64
40         cylinder 6 35.56     45.72 -45.72
41
42         cuboid  99  139.7 -139.7   139.7 -139.7  152.4 -152.4
43
44         media 3 1  1
45         media 1 1  2 -1
46         media 2 1  3 -2
47         media 2 1  4
48         media 2 1  5
49         media 1 1  6 -3 -4 -5
50
51         media 3 1  99 -6
52     boundary 99
```

SCALE Input File Extensions

These are the file extensions with which to associate any SCALE-specific editing capabilities (input parsing/validation, geometry visualization, etc.).

Sequence Declarator Foreground The foreground color of any text identified as a sequence declarator.

`17` `=mavric`

String Foreground

The foreground color of any text identified as a string.

`60`   `title="ANSI standard (1977) neutron flux-to-dose-rate factors"`

Configurations

> **Note**
>
> Configurations are opened via *file>Configurations...* file menu item

Workbench allows the user to configure and save multiple environments for individual applications to run through the Workbench with user-specified arguments.

Additionally, a runtime environment can be specified in place of system-defined variables.
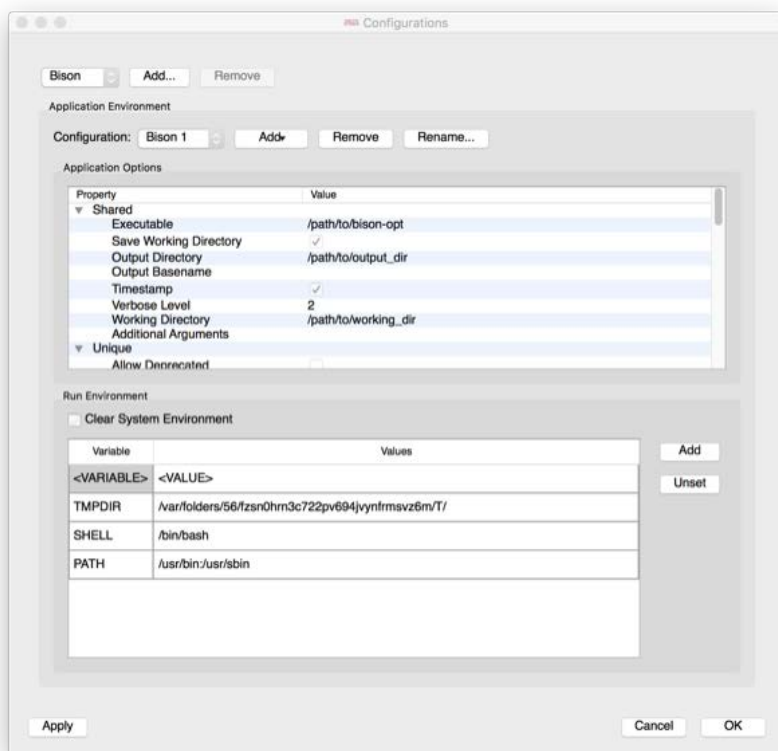
To get started, minimally,

1. Select an application for which to create a configuration. If your application is not listed, you will need to conduct the application's *Application Integration*.

2. Specify the Executable path in the application options.

Application Environment

Once an application is selected, the environment can be configured. A default configuration is provided and can be renamed and cloned. In order to run using the configuration, the executable's file path must be updated.

> Note
>
> Provided configuration names must be unique. If a duplicate is provided, it will automatically be renamed.

Application Options

This displays a list of the most recent arguments that can be passed to the application at run time. The arguments are separated into two categories:

1. Shared: Common arguments that are accepted by all available applications. The "Additional Arguments" property can be used to provide arguments as they would be entered on the command line. If an argument has been deprecated and you still wish to use it, it can be specified here.
2. Unique: Application specific arguments that will update based on selected application.

Run Environment

For each application, a list of system environment variables can be provided for use at run time. You can use any environment variables as values in the fields.

Application Integration

Below is a step-by-step guide for integrating your application into the NEAMS Workbench. Please send questions to NEAMS Workbench Help at nwb-help@ornl.gov.

1. Create an input schema for your application.
   a. If your application is MOOSE based, you may create this by running your application with the --definition flag.
   b. The schema generated with this command should support full input validation of all of your application's inputs.
   c. It should also contain autocomplete hooks that use the MOOSE templates already included in Workbench.
   d. If your application is not MOOSE based, please reference the Schema Creation section of this documentation for creating your own schema.
   e. Move the schema file to INSTALL/etc/InputDefinitions/finaloutput/YourAppName.sch, where INSTALL is the installation location of Workbench.

2. Create a grammar file.
   a. Move into the Workbench install grammar directory at INSTALL/etc/grammars, where INSTALL is the installation location of Workbench.
   b. If your application is MOOSE based, copy the grammar file from another MOOSE application, such as bison.wbg, to YourAppName.wbg.
   c. Make the name = field in your grammar file be YourAppName instead of what is currently there.
   d. Make the schema = field in your grammar file be the relative path to the schema from Step 1.
   e. The highlighter field is set up for a general MOOSE highlighter. This may be modified if you want.
   f. The templates field is set up for general MOOSE templates. These may be modified if you want.

g. You may also modify the other fields such as the tree display depth and input extensions.

h. You may learn the full scope of the grammar file fields in the Grammar Support section of this documentation.

i. Full validation and autocomplete should now be available for your input in Workbench.

3. Hook up a runtime to actually run the code on the input.

a. If your application is MOOSE based, you may simply open Workbench, go to File->Configurations, add a new MOOSE configuration, and set the `executable` field to point to your executable.

b. You may learn about the full scope of hooking up an application with runtime flags in the *Configurations* and *Runtime* sections of this documentation.

c. Your application input should now run with your executable through Workbench.

4. Create a post-processor file to scrape the output and visualize.

a. Move into the Workbench install processors directory at INSTALL/etc/processors, where INSTALL is the installation location of Workbench.

b. Copy a processor file from something of interest and modify the fields however you see fit.

c. You may learn the full scope of the processor file fields in Output Post-Processor Support section of this documentation.

d. Your application's run results should now be able to be visualized inside Workbench.

Input Support

Workbench provides a cross-platform input editing experience. Its foremost mission is to not alter the user's desired format. In fulfilling this mission, the text editor is its central feature. A significant feature set is in development that will facilitate input introspection, auto completion, and context-specific help documentation.

Current Workbench features facilitate the creation, editing, navigation, validation, and execution of input. All actions revolve around the input text file. In the absence of full component wizards, a component creator mechanism is provided for some of the components that lack obvious input definitions (compositions, geometry, etc.).

Input Creation Use File -> New file... to create an input file. The user is presented with a file dialog which allows the selection of the directory and the naming of the new input file. Upon input file creation, an empty file will be presented in a new input tab. Input editing may commence.

Input Editing The Workbench input editor is composed of several parts: the text editor, a quick navigation drop-down, validation and messages and search panels, line and column indicator, and the document navigation tree. The foremost component is the text editor, which most users will find to be similar to their favorite platform-specific text editor.

1. The text editor: uses color highlights to visually enhance the input representation. Advanced editing features include:
   - Selected text highlighting (after a word is selected, matching selections are also highlighted)
   - Column selections
   - In-line calculator

2. The quick nav: provides a drop-down list of the high-level components found in the input. This drop-down is populated once Workbench successfully parses the input document.

3. The run configurations: provide a list of applications the user has configured to be able to run from within Fulcrum. Selecting the last item, "Customize," displays a dialog in which the user may add, remove, or edit configurations to be displayed here.

4. The run menu: provides a means of running the currently selected configuration.

5. The view menu: provides a list of ways to visualize information provided in the given input. Currently only includes geometry visualization.

6. The edit menu provides a list of ways to edit the given input, including accessing autocomplete capabilities, toggling comments, and indenting/un-indenting text.

7. The search panel: provides search capability within the given document via regular expression patterns.

8. The validation panel: provides a list of parser and validation error messages. The output panel, which provides message listing from the runtime upon execution of the input, would be visible if "Messages" were selected.

4. Validation/messages selector: shows its corresponding panel above when a button is clicked.

5. Line and column indicator: indicates the line and column of the cursor. Clicking this indicator will display a line and column edit box. Entering the line and optional column (comma delimited) will navigate the cursor to the entered line and column.

9. Current input context: if the input is valid up to the cursor's current location, this displays the path of the associated parser input node.

10. Navigation panel: provides tree listing of the high level input components. This tree is populated once Workbench successfully parses the input document.

Note

You can open files associated to files listed in the navigation panel by right clicking the navigation item to acquire its context menu from which an 'Open Associated files' sub menu will list all files that are associated. Left click any of the associated files and it will be loaded.

Workbench is integrated with the parser package which provides Workbench insight into each component of the input. This coupling allows NEAMS Workbench to provide line and column quick navigation capability via the quick nav drop-down and document navigation components, and it also facilitates the integration of any geometry visualization via the new Atlas geometry package. Upon any text edit, Workbench uses the NEAMS Parser package and reprocesses the input, checking for parse errors (i.e., missing terminators or components or incorrectly typed components). After reparsing the input, the quick nav and document navigation components are updated.

Column Selection

Column selection can be used to select a rectangular area of a file. To begin the selection, press the Alt/Option key and move the cursor as you would for a standard multiline select. The selected area then allows for insertions, deletions, copying, cutting, and pasting to another location.
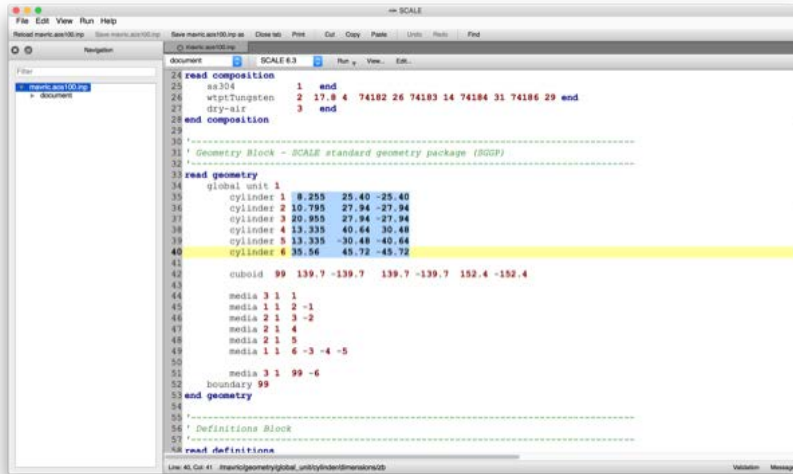
In-Line Calculator Selected functions and expressions can be evaluated by clicking Edit -> Evaluate or by using the CMD+E key combination. Expressions can contain multiple operators to be evaluated at once.

In addition to basic arithmetic operators (+, -, '*', /, ^), the following functions are available for evaluation:

sqrt, cos, sin, root, abs, min, max, avg, sum, mul, floor, ceil, exp, log, logn, log10, hyp, ifFunction, clamp, inrange, sign, deg2rad, tan, equal, acos, asin, atan, cosh, tanh, sec, csc, cot, sinh, round, roundn, d2g, g2d, r2d

> Note
>
> min, max, avg, sum, and mul are fuctions that accept a variable number of input parameters. i.e. $\min(x_1, x_2, x_3,..., x_n)$

Example Using Function

cos(pi) = cos(pi)

By selecting and evaluating the function following the equal sign, the selection will change to the evaluated term, so the example above will now read, cos(pi) = -1

Example Using Basic Operands

(4 * 2) ^ 5 / 4 + 1 = (4 * 2) ^ 5 / 4 + 1

By selecting and evaluating the expression following the equal sign, the selection will change to the evaluated term, as follows.

(4 * 2) ^ 5 / 4 + 1 = 8193

Input Component Creation

The component creation is made available as an extensible means to generate components where the components can vary from simple, one-line components like a composition or geometry region, to complex complete inputs describing entire spent fuel canisters. Instantiating the input component creator is straightforward. With the cursor at the desired insertion location, click the Edit...>Autocomplete (CTRL+SPACE key shortcut), which will list all available autocomplete input at that location in the input. Any autocompletion record with "(Configurable)" following it can be selected to instantiate that component's creator.
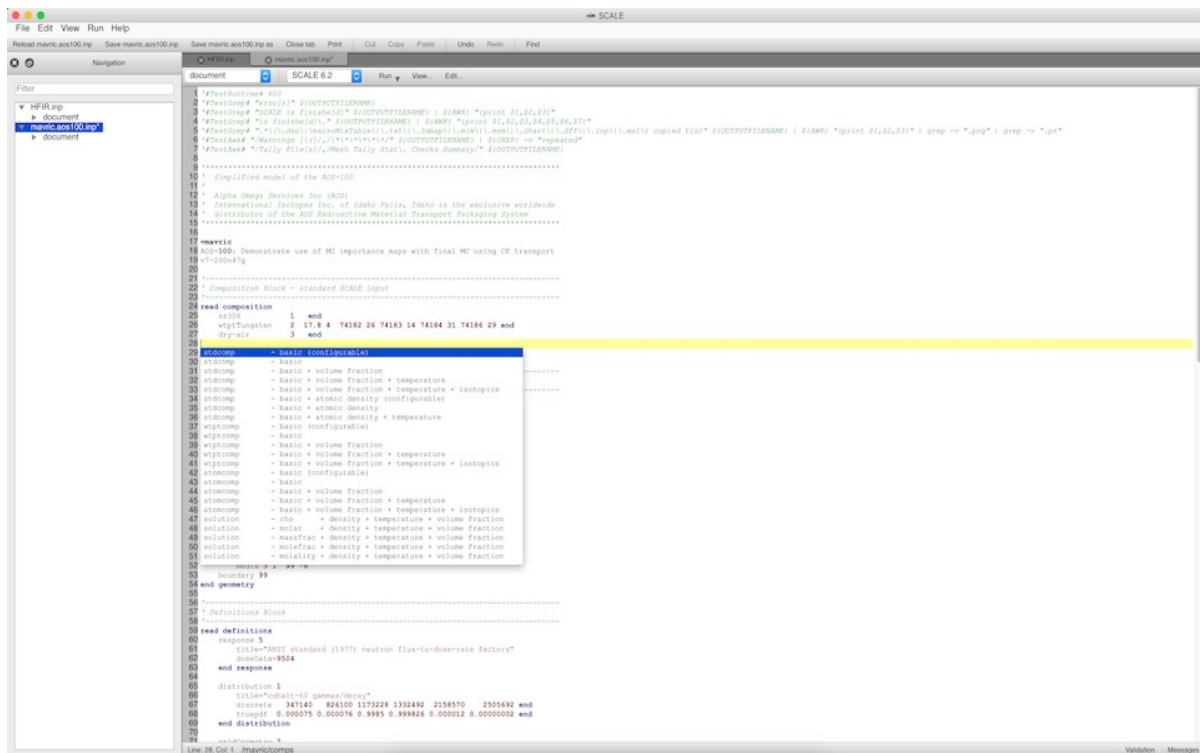
Note: Configurable autocomplete, or component creators, are not available in all contexts. If there is a component you would like to have auto-completable or configurable, let us now with an email to NEAMS Workbench Help at nwb-help@ornl.gov.

The image above illustrates the basic composition creator. Upon left clicking the OK button, the results are placed at the cursor location from which the component creator was instantiated.

Validating Input

Upon editing or initial viewing of an input file, Workbench uses an input parser and Hierarchical Input Validation Engine (HIVE) packages to process the input and determine the input's validity. Error messages from the parser and HIVE packages are placed into the validation panel for the given input. The types of validations performed by HIVE are numerous and beyond the scope of this document. The HIVE package attempts to communicate in as meaningful a way as possible. As always, if an error is ineffectively communicated or not communicated at all, please contact NEAMS Workbench Help at nwb-help@ornl.gov.

The above image illustrates a common mistake in which composition of h2o (H Two O) has been specified as h20 (H Two Zero). The validation panel lists the mistake as an error, specifying the input file path, the line and column (line 56, column 2), the name of the input component (*decl*, an abbreviation for *declarator*), and the value. The validation panel also states that the value is not a member of the enumerated set of composition names, and it provides a closest match list of what could have been intended, one of which is the correct h2o. This error message can be clicked, and the input editor will automatically navigate to the location of the error (line 56, column 2), facilitating fast response time for the user.

It is important to note that while a significant amount of NEAMS has validation checks enabled, some components are too complex for HIVE to validate. For example, geometry errors involving undefined or doubly defined regions are not validated. These more complex validation procedures are performed by their respective components in NEAMS. In the case of geometry, these errors are communicated upon geometry visualization via the respective geometry package.

Input Navigation

Navigation of input can be done several ways. The simplest is via the regular cursor movements or scrollbar manipulation, as would be done in any text editor. Depending on the actions (error look up, document searching, etc.) this can become unproductive when dealing with large or unfamiliar inputs.

Workbench provides the document quick nav drop-down at the top of the input editor. The drop-down menu lists the high level components of the input. Clicking one of these items will place the cursor at the start of the component. If the component is not within the current scroll view, the scroll bar is automatically manipulated to ensure that the component and the cursor are visible.

The document navigation panel provides the same capability as the quick nav drop-down.

Beyond the document navigation components mentioned here, the line and column indicator can be clicked to show a line and column edit box where the desired line and optional column number can be specified to quick nav to the specified location. This line and column indicator can facilitate jumping to document locations mentioned in warnings, errors, or other contexts.

As the NEAMS input infrastructure is modernized, more capabilities will be added. The latest capability is the ability to "Go to Definition." Upon right clicking in an input file, a context menu is

displayed, the first item of which will be "Go To Definition of ..." which allows the user to quick nav to the definition of the component. If the component is related to another input component (e.g., a geometry unit media record's mixture is defined in the composition block's composition records), then left clicking the "Go to Definition ..." will place the cursor on the component and select the text at that location. If the component is defined by more than one related item (e.g., multiple composition records defining a single mixture), then a dialog will be presented to allow you to select a part of the definition to navigate.

Saving Input

Saving input is straightforward. The input file must be selected in the document navigation panel, after which clicking File -> Save or File -> Save As... will save the file.

Alternatively, when the input file is selected, the CMD+S key combination will save the file.

Note

Workbench will conduct auto saves of your text files as filename.fulcrum.autosave. If Workbench quits at any time while the file is modified, then the filename.fulcrum.autosave acts as a recovery file. If Workbench successfully shuts down then this file is removed.

Executing Input

The execution of input can be conducted by clicking the Run button, or this can be accomplished by having the input file selected in the document navigation panel and conducting the CMD+R key combination, which will launch runtime configuration with the selected file. Upon invocation, a new tab is added to the messages panel, and messages from the selected runtime configuration will automatically be listed.

The execution can be stopped by right clicking the message panel to acquire the message panel's context menu, which contains the "Kill" and "Terminate" menu items. Terminating the execution will give the job a chance to gracefully exit, cleaning up working space and deleting temporary files. Killing the execution will be immediate and will leave temporary files lingering.

> **Note**
>
> Depending on the operating system, killing or terminating your execution may leave stray temporary files remaining, and orphaned child processes may still be running. It is advised to (1) check the temporary directory for stray files to avoid unwanted disk space consumption and (2) to use your operating system's process monitor to ensure that potentially orphaned child processes are no longer consuming your machine's processors.

Beyond the runtime execution of the input, the Vulcan material processor can be invoked via the Run>Mixing table menu item for material review, and the messages panel will be populated with the mixture pool table listing the mixture, nuclide, and nuclide number density triplets. Additionally, the input components can be listed in a directory style listing via the Run>Input Listing menu (or toolbar) item.

Geometry Visualization

Geometry visualization uses the new Atlas package and corresponding Atlas view component in Workbench. Currently, only SCALE geometry is supported. Units are centimeters and as such all geometry units are displayed in centimeters.

Getting Started

To visualize geometry that is supported, load the input file.

> **Note**
>
> Geometry visualization is only available if the document was parsed successfully.

Click the "View" button on the toolbar, and then click the "View geometry" item in the menu that pops up.
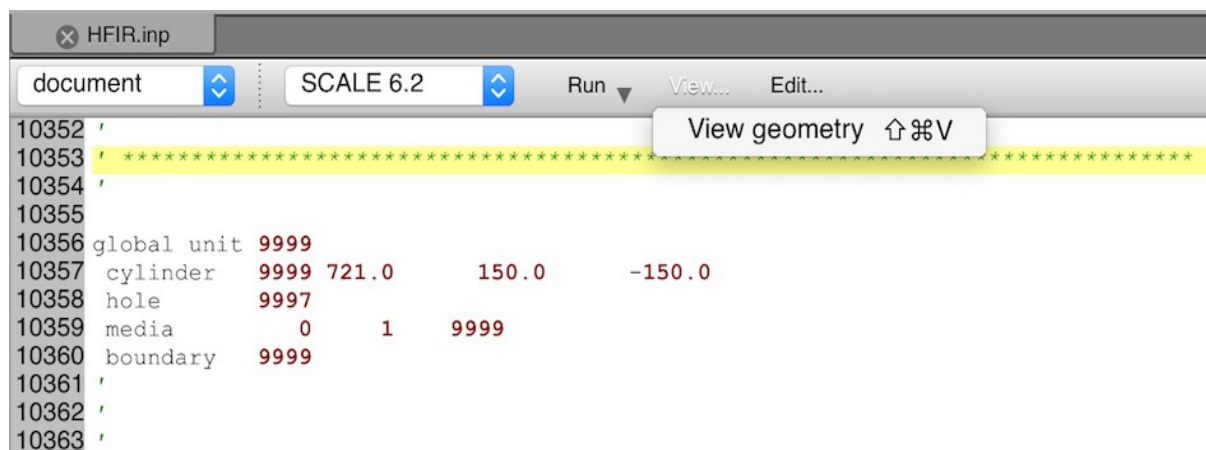
A new panel will open in Workbench with the initial Atlas geometry view loaded. In the initial Atlas view—the top (X-Y)—X increases from left to right and Y increases from bottom to top.
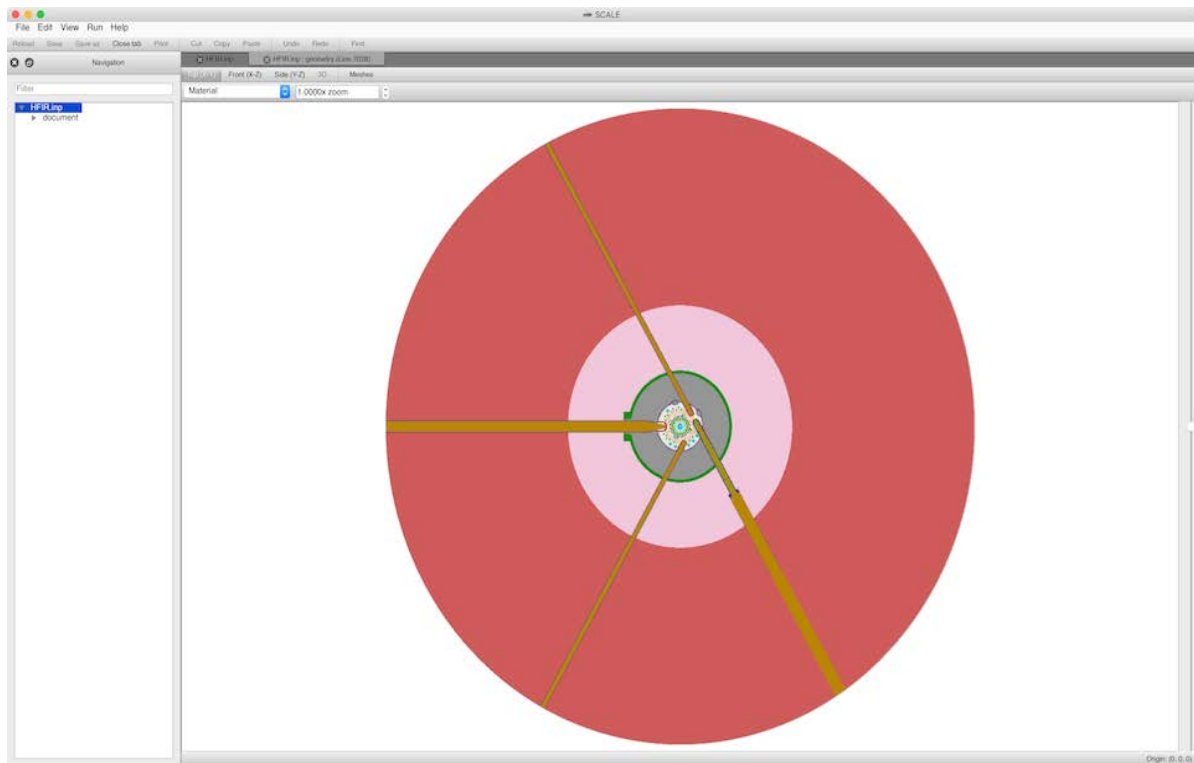
Atlas Geometry Package

The Atlas geometry package is part of the SCALE modernization effort. Specifically, it is a re-usable C++ package that can interpret SCALE geometry (NEWT, KENO V.a. and VI). Workbench uses Atlas to present geometry visualization and allow geometry interaction.

Atlas Views

The Workbench Atlas component currently supports the three primary Cartesian cut planes, specifically the top (X-Y), the front (X-Z), and the side (Y-Z). Each view is accessible via its respective view-mode buttons. It should be noted that 3D is not yet implemented. Each view has an axis slider on the right side of the view. The axis slider controls the elevation of the cut plane in the current Atlas view. Moving the slider up increases the elevation, and conversely moving it down decreases the elevation. Atlas conducts dynamic ray tracing of the implicit geometry equations composing the SCALE geometry. As a result, some visual artifacts may be observed when zoomed to a level of less than 1e-9 cm.

The screenshot below shows the front (X-Z) view. X increases from left to right, and Z increases from bottom to top.
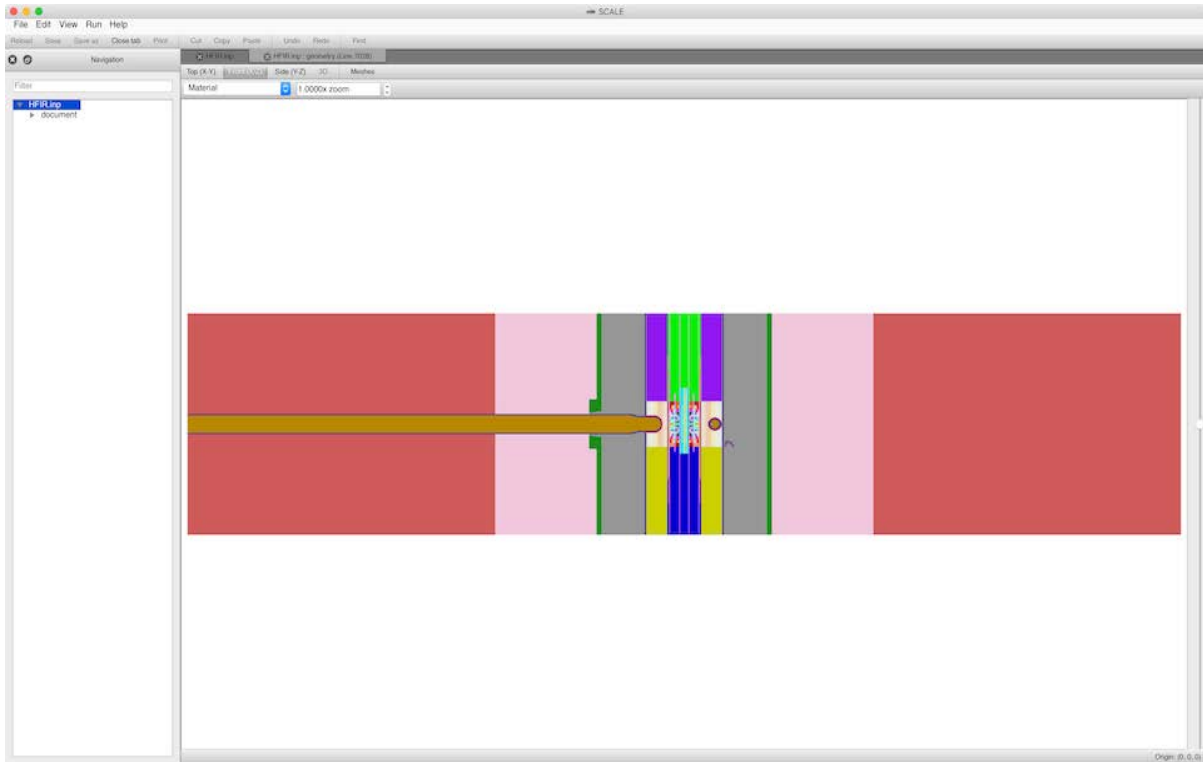
The screenshot below shows the side (Y-Z) view. Y increases from left to right, and Z increases from bottom to top.
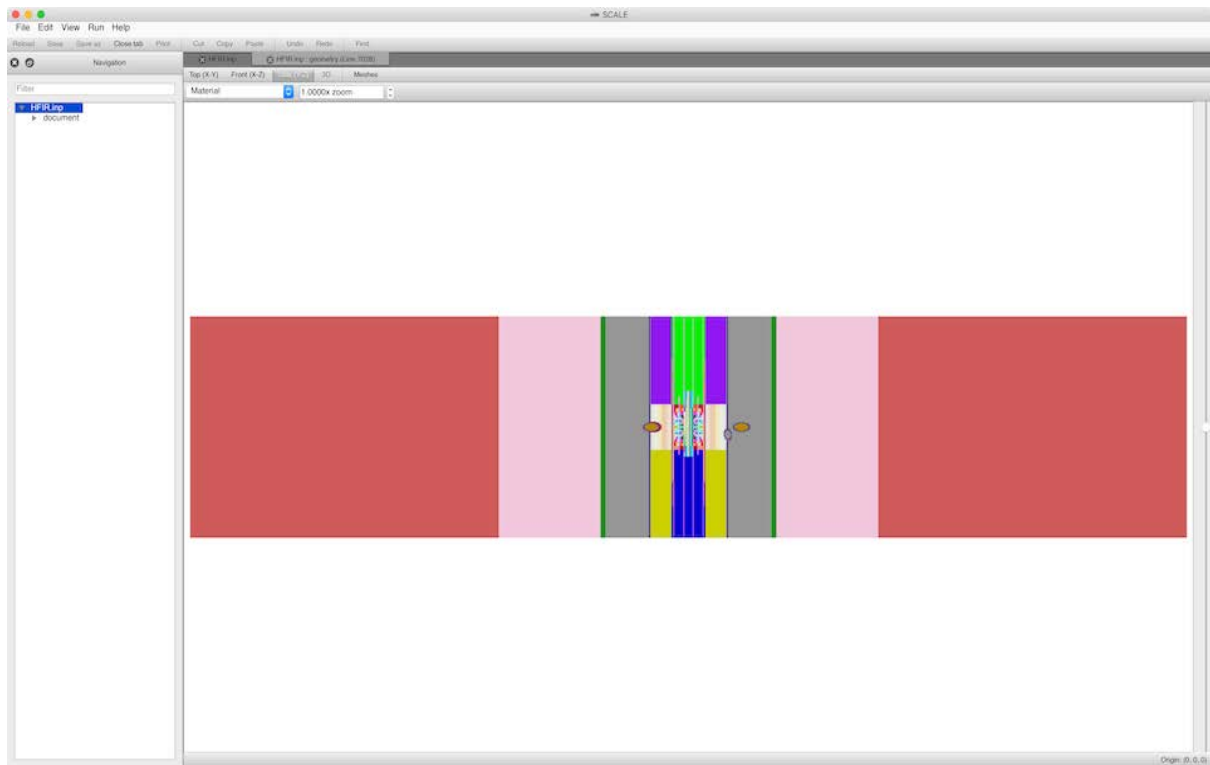
Each view has a "View origin" label which indicates the origin of the view in absolute Cartesian coordinates.

Rendering Modes

Multiple geometry rendering modes are available to the user.

Material

This renders the geometry by filling in the bodies represented by the units' media definitions with their own respective colors.
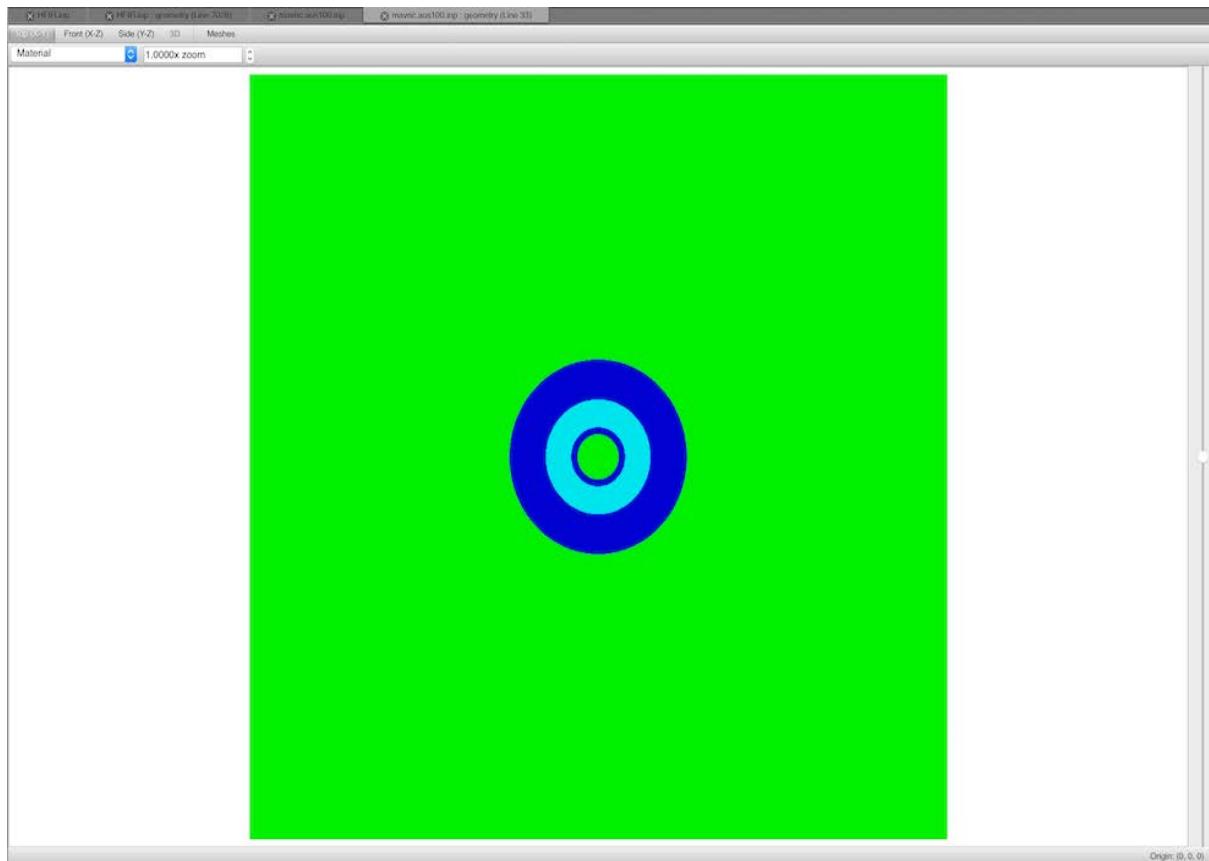
Material + Outline

This renders the geometry as in "Material," but it also draws (in black) the outlines of the individual geometry regions that compose the materials.
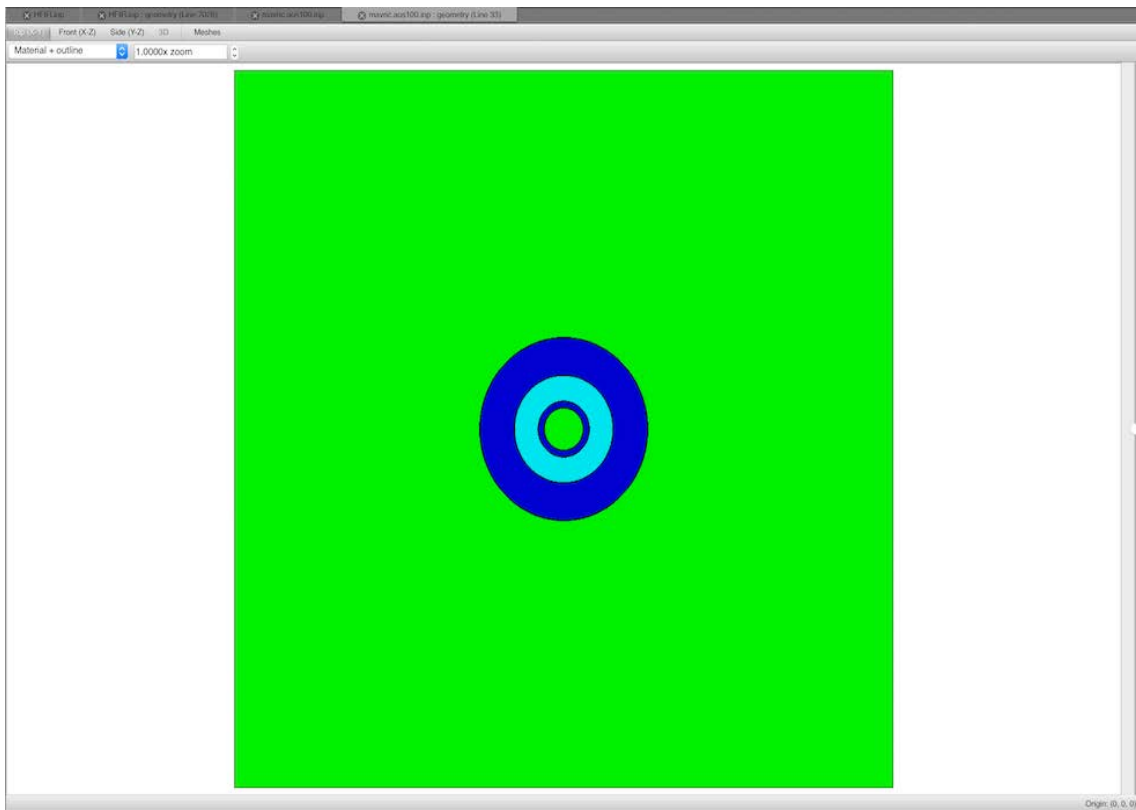
Outline

This shows (in the associated materials' colors) only the outlines of the units' regions.
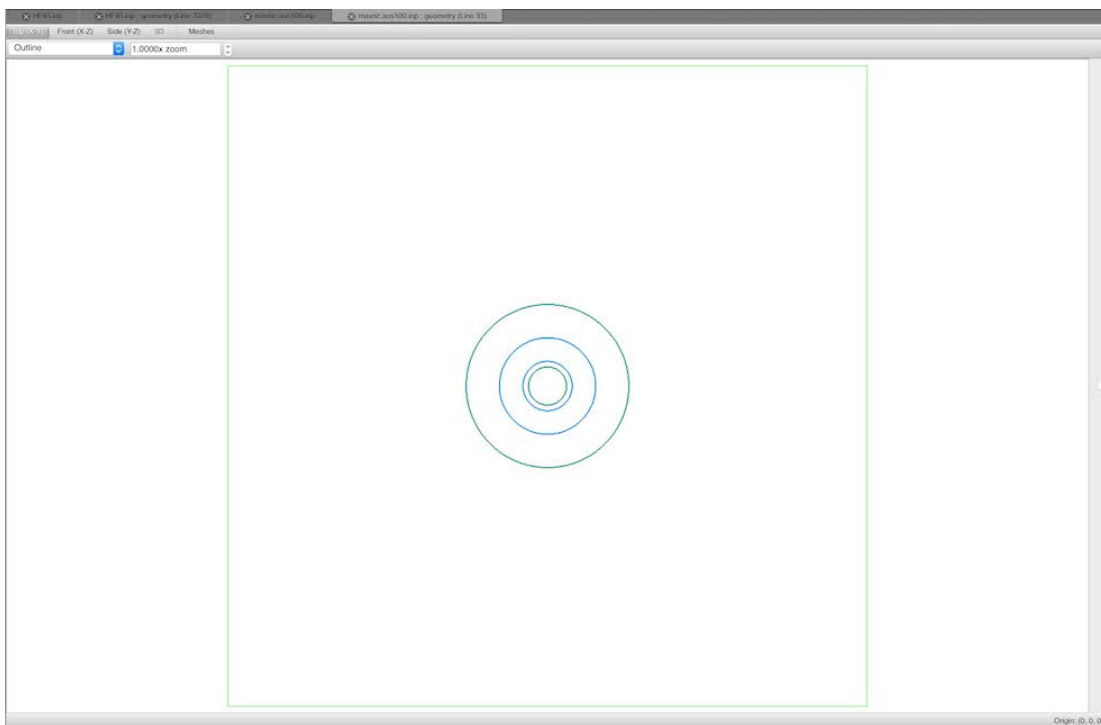
Overlay

This renders (in black) the outlines of the units' regions and overlays any mesh data the user has selected in the mesh browser.
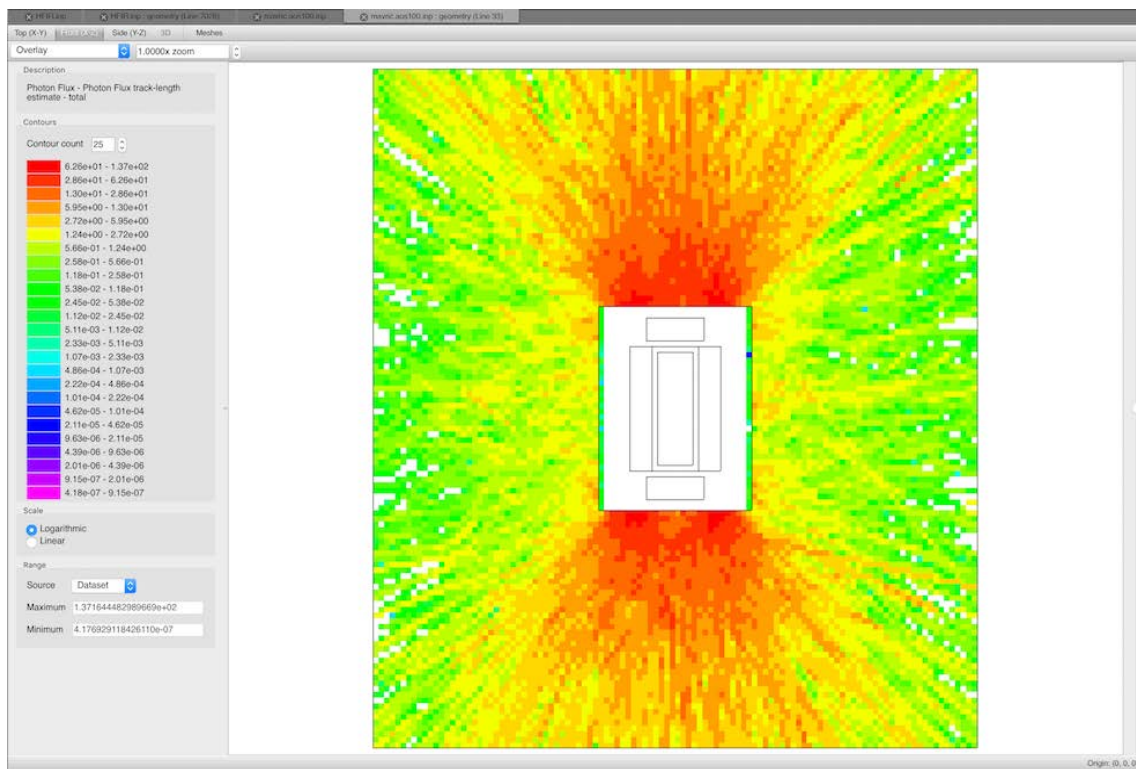
Overlay + Boundaries

This renders the geometry as in "Overlay," but it also draws the boundaries of the meshes specified in the data selected by the user in the mesh browser.
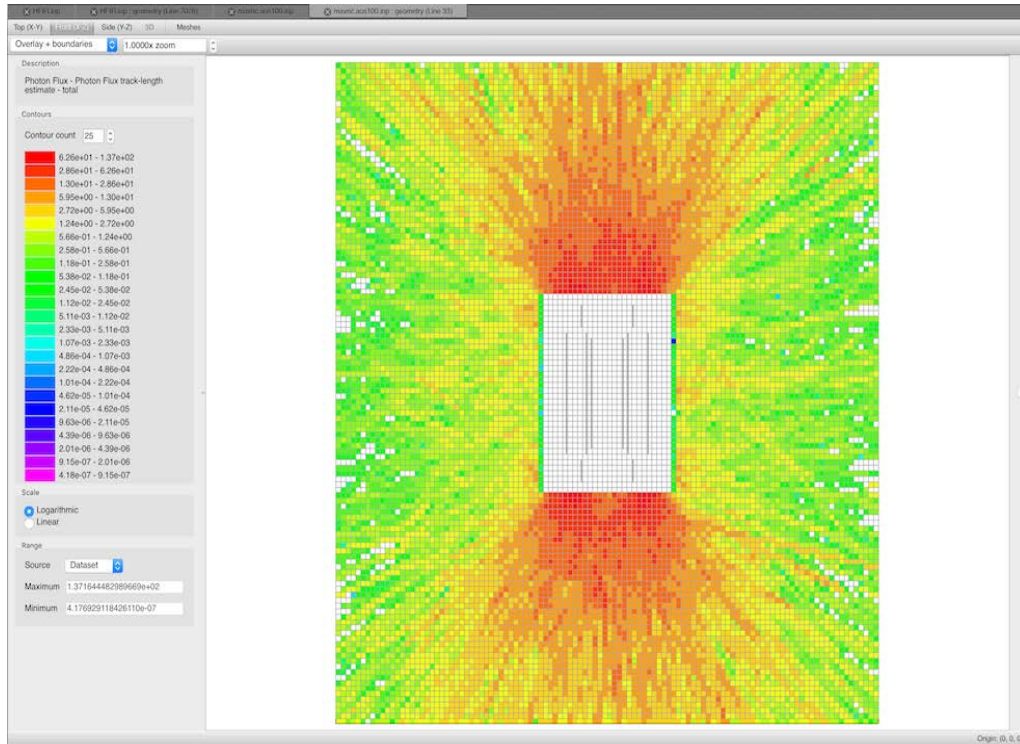
Note:

Only Cartesian mesh boundaries are supported for this release. Cylindrical mesh boundaries will be supported in a future release.

Origin Crosshairs

The origin can be indicated with crosshairs by toggling the "Show view origin" button.
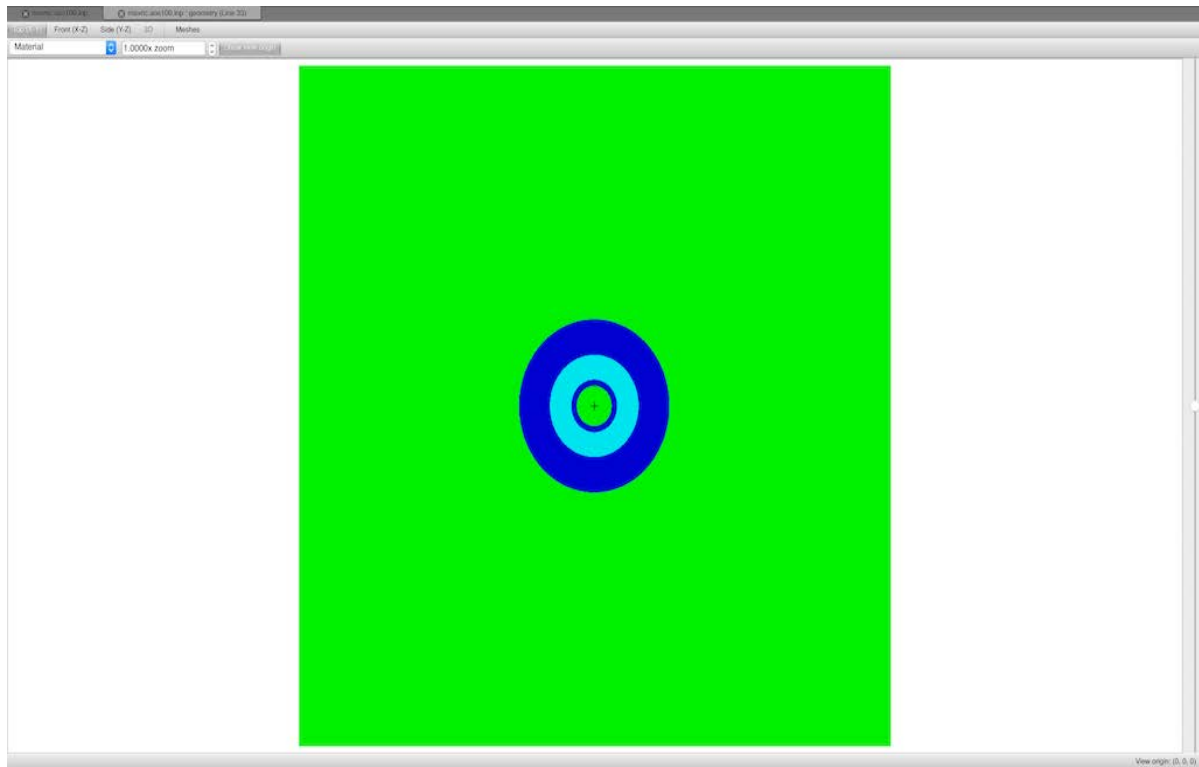
Panning

Panning is performed by holding the left mouse button and the SHIFT key and dragging in the opposite direction of the desired pan. Upon initial click, the location under the mouse will be the same location upon concluding the pan. That is, if you leave click and drag to the lower right corner, the upper left corner will be the new center, assuming a rectangular image.

Recentering

Often there is a point of interest that must be the center of your view. A simple double click on the point of interest will recenter the view.

Additionally, clicking on the "View origin" label in the lower left corner of the Atlas view will present the user with an edit box. Typing two comma-separated values into the edit box will recenter.
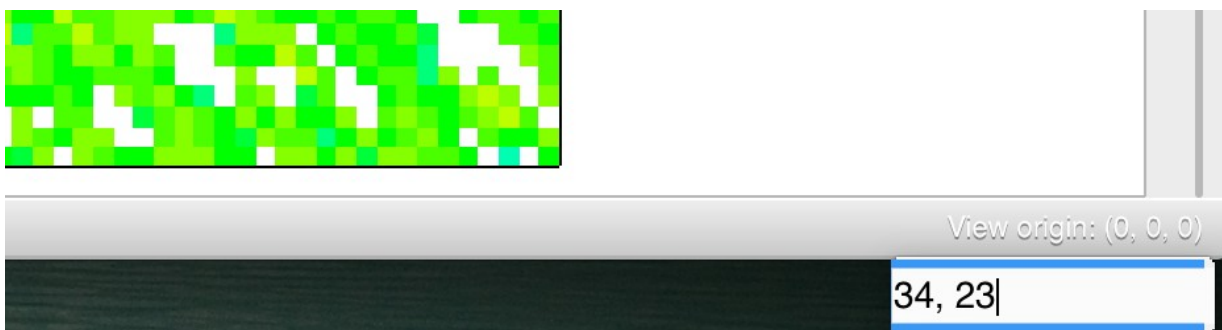
This "View origin" edit box works in 3D, as demonstrated in the following:

- Entering a single value changes the elevation of the cut plane axis, which is equivalent to using the slider in X-Y, Z as specified.

> • Entering two values performs a pan in the view. That is, specifying 0,12 in the X-Y view pans to the location X=0, Y=12.
> • Entering three values performs both an elevation change and a pan.

Zooming

There are two ways to zoom. The most obvious is to use the zoom factor spin box that zooms in or out of the center of the view.

The second means of zoom is to use the mouse to draw a zoom reticle by left clicking and dragging from the top left reticle corner to the desired bottom right corner. Upon release of the left mouse button, the zoom will be implemented. If the user decides that while drawing the zoom reticle, the zoom is no longer desired, then pressing the ESC key will terminate the zoom action.
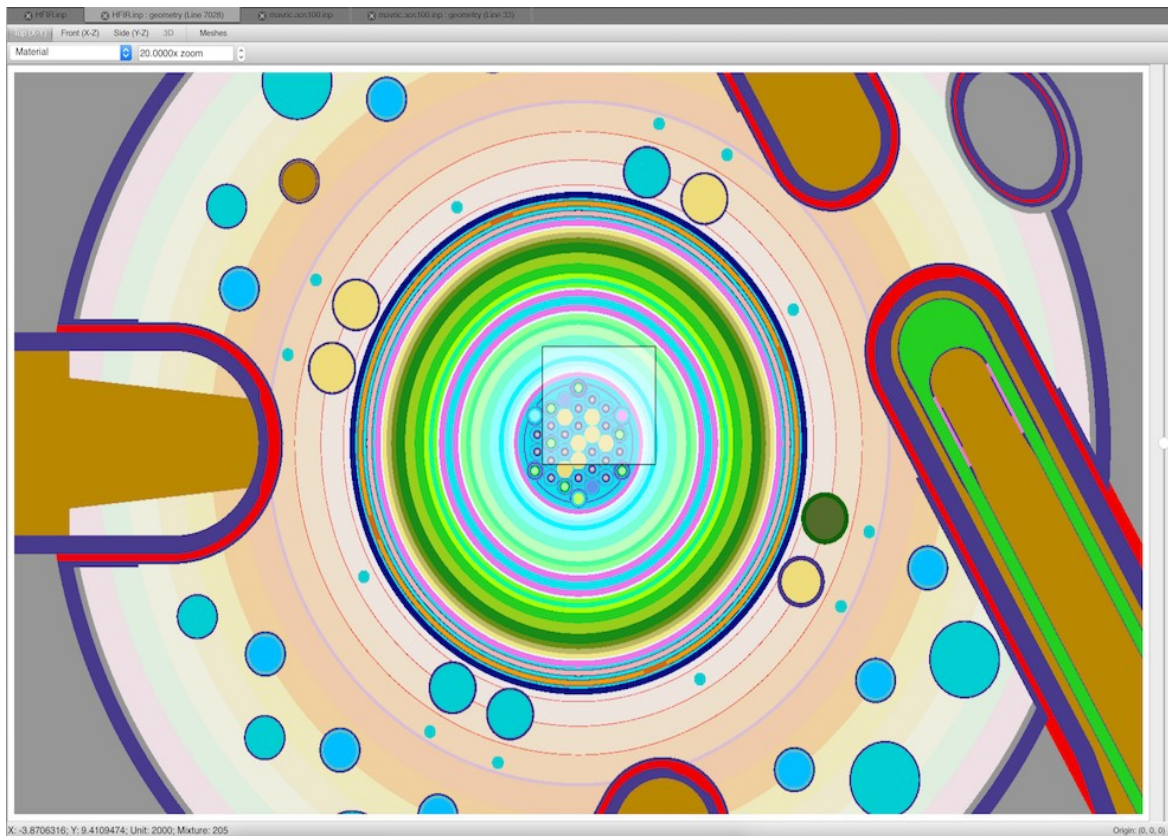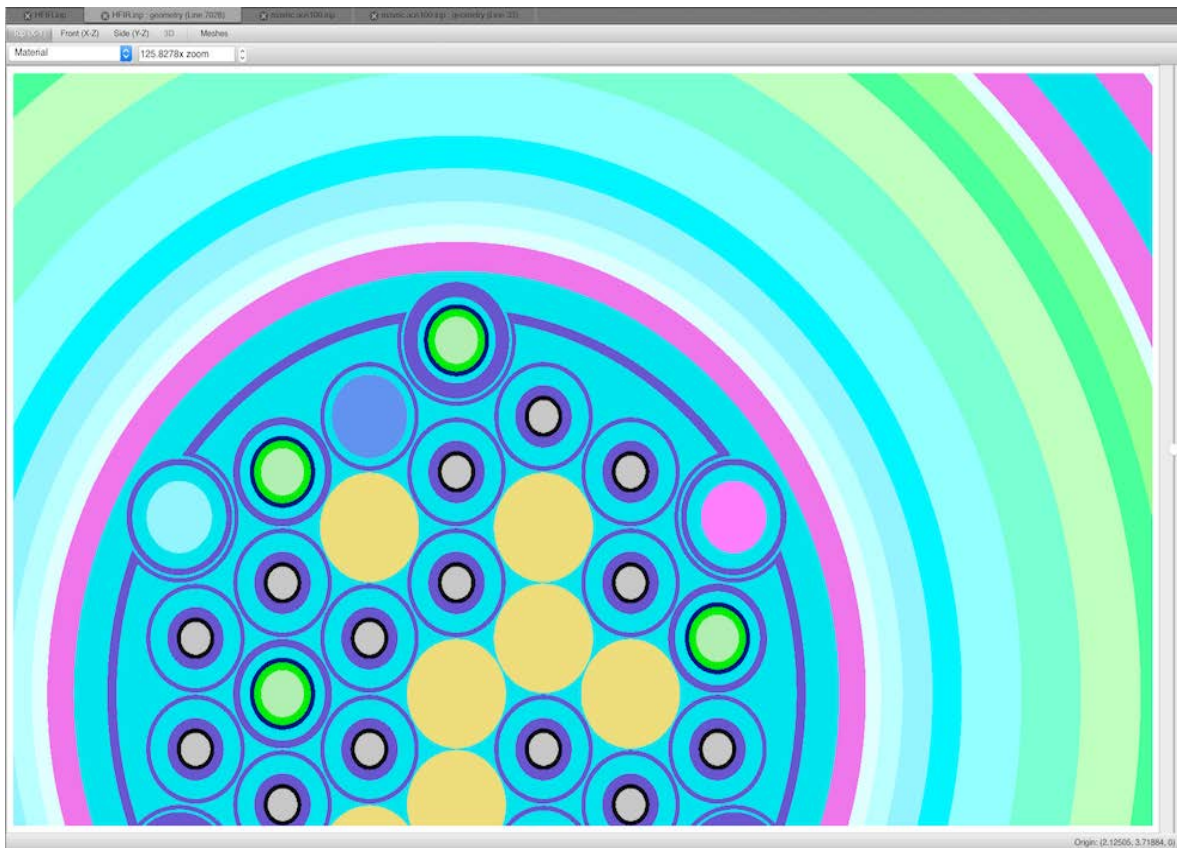
Zoom-to-Fit

To zoom out to the extent of the model, simply left click, hold, and drag up and to the left, and then release the mouse button. The view will fit the entire geometry into the available Atlas view panel.

Grammar Support

One of the major goals of Workbench is to support/integrate multiple modeling and simulations toolsets. Each toolset may have its own input format, and users will expect to be able to open/edit inputs native to their respective tools of choice. To facilitate this capability, Workbench supports what has been dubbed "grammars," or files that provide information to assist with supporting these native input formats. These grammar `*.wbg`, files are expected to be in the SON format, with fields describing the following:

- Input parser, schema, and validator
- Templates to assist with auto-completion
- Patterns used for syntax highlighting
- Other metadata to assist with editing content, displaying document hierarchies, etc.

The following is an example grammar file:

```
commentDelimiter = "" enabled = true

extensions = [inp in input] highlighter = "highlighters/scale.wbg" maxDepth = 2

name = SCALE parser = SCALE

schema = "../InputDefinitions/finaloutput/scale.sch" templates = "../Templates/autocomplete"

validator = SCALE
```

Grammar files provided by Workbench are expected to be found here:

/path/to/install/etc/grammars/*.wbg

Users may create their own grammar files to be used the next time Workbench is launched. Those files are expected to be found here: /path/to/user/home/.workbench/6.3/processors/*.wbp

Note: When creating custom grammars, if the name is identical to one deployed with Workbench, then the user-defined grammar overrides the other. If the user creates multiple grammars with the same name, then the last-seen grammar is used:

In this example, there are two `Grammar 1`s, `grammar1.wbg` and `grammar3.wbg`. The one defined by `grammar1.wbg` is discarded, replaced by the one defined by `grammar3.wbg`.

```
grammar1.wbg:

...

name = "Grammar 1"

...


grammar2.wbg:

...

name = "Grammar 2"

...
```

Input Parser, Schema, and Validator

Input Parser

The `parser` field is used to specify the type of parser to be used when processing a given text file's contents. Workbench natively supports several SCALE-specific input formats that can be specified here:

- `ChartPlot`: SCALE's parser to process ChartPlot files (defaults to `*.chart`)
- `PtolemyPlt`: SCALE's parser to process Javapeno's plot files (defaults to `*.ptp`, formerly `*.plt` in Javapeno and older versions of SCALE)
- `SCALE`: the main SCALE parser (defaults to `*.inp` ,``*.in``, and `*.input`)
- `SON`: SCALE's parser to process SON files (defaults to `*.son`)

With the addition of the Workbench Analysis Sequence Processors (WASP) library, Workbench also supports the following parsers:

- `DDI`: "definition-driven interpreter", a schema-informed parser created to support Dakota
- `GetPot`: the input format used by MOOSE and MOOSE-derived codes
- `JSON`: JavaScript Object Notation, a storage/data exchange format used extensively in websites and increasingly in mobile/native apps

WASP also provides an improved, more efficient implementation of SON that can be enabled via `waspSON` to differentiate it from SCALE's SON.

> **Note**
>
> SON originated within SCALE, as did some file types that use SON for their storage format, like CustomPlot (`*.spf`) files. Any such SCALE-SON–dependent files will list the parser as `SON`.

Input Schema

> **Note**
>
> Schema (`*.sch`) files are assumed to be SON-formatted, and the SON parser, whether SCALE or WASP, is selected based on whether the associated `validator` is SCALE- or WASP-based.

Input Validator

The `schema` field is used to specify the path to the schema, which is the file used to describe the input format. Since some text file formats will not have an associated schema, this field is optional.

The `validator` field is used to specify how to validate the given text file against its associated schema. This field's value should be `SCALE` or `WASP`, depending on the `parser`. This field is optional for the same reason that `schema` is optional. Though encouraged, document validation is not required if a schema is available.

> **Warning**
>
> If a schema is not available, this field should not be included.

Templates for Auto-Completion

The `templates` field is used to specify the path to the directory containing templates to aid in auto-completion for the given text file. These templates are placed in the file at the line/column at which the user triggers auto-completion logic. These templates can allow the user to quickly populate an input file with various blocks of text that may be needed but are not familiar to the user, or it can provide an example by which users may learn the type of information to provide for a given field or block of text. Since some code developers may not provide templates to aid with auto-completion, this field is optional.

Syntax Highlighting

The `highlighter` field is used to specify the path to an associated highlighter (`*.wbh`) file. These files provide a series of syntax highlighting rules to help users visually identify pieces of their text files, such as strings, numbers, comments, etc. Since some code developers may not provide highlighter rules, this field is optional.

Highlighter File

Highlighter files are SON-formatted files that provide a series of named `rule` objects containing the following fields:

- `background`: the color of the line behind the matching text
  - `alpha`: the alpha, or transparency, channel (0-255, where 0 is transparent and 255 is opaque)
  - `blue`: the blue channel (0-255): blue = 0, 128, 255

- – `green`: the green channel (0-255, ...): green = 0, 128, 255
- – `red`: the red channel (0-255, ...): red = 0, 128, 255
- `bold`: whether the matching text should be given a **bold font weight**
- `foreground`: the color of the matching text (see `background` for channel descriptions)
- `italic`: whether the matching text should be *italicized*
- `pattern`: the regular expression pattern used to identify sections of text to which to apply the given style

Note

Workbench is based on the Qt framework, which has native support for regular expressions documented here. Specifically, the patterns should follow the RegExp syntax and are treated as case insensitive (e.g., "test pattern" and "TEST PATTERN" are treated the same).

Highlighter files are expected to be found here:
`/path/to/install/etc/grammars/highlighters/*.wbh`

The following shows a SCALE-style comment rule, a double-quoted string rule, and an integer rule:

```
rule("Comment") { background {

alpha = 24

red = 0

green = 128

blue = 0

}

foreground {

red = 0

green = 128

blue = 0

}

italic = true

pattern = "(^'|%|//).*"

}
```

```
green = 0

blue = 0

}

pattern = '"[^"]*"'

}

rule("Integer") { background {

red = 0

green = 0

blue = 128

}

foreground {

red = 255

green = 255
```

> **Note**
>
> Rules are applied in the order they are defined. The following two lines demonstrate how the above rules work: ' comment before the string "inside a double-quoted string" comment after the string
>
> "inside a double-quoted string. % starts a SON comment."
>
> The following shows what happens if an integer is found in general text and inside a quoted string: Four score and seven (87) years ago...
>
> "Four score and seven (87) years ago..."

Other Fields

`commentDelimiter` specifies the string to be used when (un)commenting code via the keyboard shortcut or "Toggle comment" action.

`enabled` indicates whether the grammar should be enabled for use in Workbench.

`extensions` is a list of file extensions with which to associate the grammar. If a given input file's extension is found in a single grammar's list of extensions, it will automatically be enabled for the text file. Otherwise, no grammar is applied, but the user may still manually select one via the grammar drop-down box.

> **Note**
>
> If a grammar is disabled, its extensions are not checked when trying to determine whether a given text file has any associated grammars.

`maxDepth` is how many levels past the document root are displayed when populating the navigation tree. See the following example of a very simple SON document:

```
obj1 {
array [1 2 3] key1 = value1 obj2 {
key2 = value2
}
}
```

Its corresponding navigational hierarchy with a *maxDepth* of 2:

```
input.son
document
obj1
array key1 obj2
```

Its corresponding navigational hierarchy with a *maxDepth* of 3:

```
input.son
document
obj1
array
value value value
key1
value
obj2
```

`name` uniquely identifies the grammar and allows the user to know which grammars are available when editing text files.

Runtime Requirements

This section documents how the NEAMS Workbench runtime environment works and can be extended.

     • Python. Python 2.7.8,10 tested.

Runtime Environment Basics

The purpose of the runtime environments is to buffer the user from the nuances of running a supported application. The user does not need to know whether the target application requires the user to run it from the directory where a given input file resides, and the user does not need to manually copy desired output from the working directory to the output directory. These details should be handled by the runtimes; the user only needs to know how to run the scripts and that it will work.

Base Runtime Environment workbench.py

`workbench.py` is the base runtime script and contains the majority of the execution logic. It also provides a set of "shared" options that any script that extends it will inherit; run the following to see the full list:

```
python workbench.py --help
```

Execution Stages

Pre-run

This stage is when the runtime might create directories it needs for execution, such as the working directory (if it does not already exist).

Run

This stage is when the actual execution occurs, running the given executable, capturing/printing its output, and reporting the result (return code) of the execution.

Post-run

This stage is when the execution has completed and the runtime might need to perform some extra housekeeping tasks, such as copying files from the working directory to the output directory, removing some extraneous files that may have been generated outside the working directory (depending on the executable's own behavior), etc.

The following is the simplest example of how a user might run a given executable via the runtime script:

```
python workbench.py -e /path/to/exe -i /path/to/input1
```

This performs the following:

- Creates a working directory in which to run the executable (defaults to a new directory in the user's temp directory, `/tmp/<app>.<user>.<process>`)
- Changes the working directory to `/tmp/<app>.<user>.<process>`
- Runs the executable
- Changes the working directory to the previous working directory
- "Cleans up" by removing `/tmp/<app>.<user>.<process>` in its entirety (including any files that were generated during execution)

The following is a similar example, but now it overrides the working directory:

```
python workbench.py -e /path/to/exe -i /path/to/input1 -O /home/user/work_dir
```

This performs the following:

- Creates the given working directory in which to run the executable:
  `/home/user/work_dir/<app>.<user>.<process>`
  Changes to /home/user/work_dir/<app>.<user>.<process>
- Runs the executable
- Changes to the previous directory
- Since the output directory was explicitly provided, it will not be cleaned up. This is because the runtime treats a custom working directory as a flag to indicate it should not be removed after execution is completed.

Creating a Runtime Environment (Extending workbench.py)

Application developers may wish to create a runtime environment specific to their applications. Ideally, all that a developer would need to do is provide the application name, a list of the options the application supports, and a way to map the runtime's options to the application's options.

Application Name

This is currently used to help identify the running application that is creating the working directory or logging information about any actions the script is performing. The base script identifies itself as `workbench`, so a custom runtime for an application called `mytestapp` would define the following function:

```
def app_name(self):
"""returns the app's self-designated name"""
return "mytestapp"
```

Supporting Application Options

Runtime-supported applications likely will have their own options they support. To fully support said applications, the runtime must be able to interpret the options and pass them properly to the application.

If the runtime must support an option whose flag conflicts with one of the base script's options, it must provide a unique flag instead. For example, if `mytestapp` supports a `-e` flag that must be exposed to the user, it must be renamed to something that does not conflict with the base script's options, such as `-E`.

Listing Supported Options

The base script provides a method that derived scripts can override to return a list of `dicts` containing metadata describing the supported options. Since the base script uses Python's `argparse` module for processing supported options, the `dicts` may contain the following fields (in alphabetical order):

- `action`: the action to take when this option is specified. This is mostly pertinent to Boolean flags, whose presence indicates `True` or `False` values. It can also be used to provide access to a custom action class if the developer is familiar with `argparse.ArgumentParser`
- `default`: the default value to use if the option is not specified
- `dest`: the variable in which to store the parsed result
- `flag`: the identifying flag, such as `-e`, `-i`, etc.
- `help`: the message to print next to the flag in the `--help` listing
- `metavar`: the pattern used to indicate a value's presence in the `--help` listing. For example, `-e`'s metavar is executable: -e executable   Path to the executable to run
- `name`: the string used by NEAMS Workbench when listing the supported options
- `nargs`: the number of values expected to follow the flag
- `required`: indicates whether the option is required
- `type`: the expected data type for the option's values currently supported: `bool`, `float`, `int`, `string`, `stringlist`

Most of these values are passed as-is to the `argparse.ArgumentParser`, but the following are special cases:

- name: not supported by argparse.ArgumentParser
- `type`: the aforementioned list of supported values is for NEAMS Workbench; the value is converted to the corresponding Python type before being handed off to `argparse.ArgumentParser`

To provide the base script with a list of supported options, the custom runtime must define the following function:

```
def app_options(self):
"""list of app-specific options"""
opts = []
...
```

The following is a simple example of supporting a $-V$ option that asks the application to print its version and quit (if it works that way):

```
# assuming self.print_version is a boolean field defined in the class


def app_options(self):
"""list of app-specific options"""
opts = []


opts.append({
"action": "store_true", "default": self.print_version, "dest": "print_version", "flag": "-V",
"help": "Print the version number and quit", "name": "Print Version",
"type": "bool"
})

```

Passing Supported Options to the Application

Supporting options is the first step, but now the options need to be passed to the executable. The custom runtime must define the following function to do so:

```
def run_args(self, options):
"""returns a list of arguments to pass to the given executable"""
args = []
...
```

The following example maps the $-V$ flag from the previous example to the executable's corresponding option:

```
def run_args(self, options):
"""returns a list of arguments to pass to the given executable"""
# build argument list args = []


# print version info if self.print_version:
args.append("-V")

```

The following example maps the $-V$ flag from the previous example to the executable's corresponding option if it is not $-V$:

```
def run_args(self, options):

"""returns a list of arguments to pass to the given executable"""

# build argument list args = []


# print version info if self.print_version:

args.append("--print-version-and-quit")


return args
```

Testing

Testing is paramount. New features require new tests to be added. Execution of tests is performed by invoking the following command at the root directory: `python -m unittest discover -v`

Output Post-Processor Support

The modeling and simulation toolsets that Workbench is in development to support can produce a large amount of output, from general debugging information, to progress/status updates, to the final result of all the calculations the codes perform. Some of this output is in files that are easily accessible via stable APIs, like HDF5, but more often it is found in mixed-format (free form, tabular, etc.) text files with no simple means of extracting/visualizing. This is where Workbench's post-processing capabilities aim to fill the gap.

Users of these codes often are familiar with scripting tools/techniques to scrape the output and reformat/import it to something like Excel for plotting. Workbench has the ability to run the same sequence of commands the user would write in this situation, process the results, and create/open a resultant plot file. The commands to run, the information needed to know how to process the results, and metadata used to automatically associate these commands with given text files are stored in files and available to the user within Workbench.

These post-processor, `*.wbp`, files are expected to be in the SON format, with fields describing the following:

- Text post-processors
- Conditionally enabling post-processors
- Organizing post-processors for use in Workbench

The following is an example post-processor file:

```
% only files with the following extensions are supported

extensions = [out]
```

```
% file must contain this grep-supported pattern
filter_pattern = "best estimate system k-eff"


% place these processors in this organizational 'hierarchy'
hierarchies = ["SCALE/KENO"]


% processor to run against a given file
processor("Best Estimate K-Effective + Uncertainty") {
% delimiter to use when parsing the output
delimiter = " "


% logic to extract data from the file
logic = """${GREP} "best estimate system k-eff" ${CURRENT_FILE} | ${AWK} "{print NR, $6,


% plot series to create
scatter("Best Estimate K-Effective + Uncertainty") {
% series keys (x-values)
keys = "a:a"


% series values (y-values)
values = "b:b"


% series value uncertainties (+/- unc)
values_uncertainty = "c:c"
}
}
```

Post-processor files provided by Workbench are expected to be found here:

/path/to/install/etc/processors/*.wbp

Users may create their own post-processor files to be used the next time Workbench is launched.
Those files are expected to be found here:
/path/to/user/home/.workbench/6.3/processors/*.wbp

Text Post-Processors

Post-processor files contain a list of `processors` that define a command to execute and how to
process the results of the command. The first thing to note is the processor must be named. The user
must provide a unique identifying string as follows: `processor("Processor name")`. This
name is used to identify the post-processor in Workbench so the user knows which processors are
available.

> Note
>
> When creating custom post-processors, if the name is identical to one deployed with Workbench, then the user-defined post-processor overrides the other. If the user creates multiple post-processors with the same name, the last-seen post-processor is used:

```
1: processor("Processor 1") {...}

12: processor("Processor 2") {...}

23: processor("Processor 1") {...}
```

> In this example, there are two `Processor 1`s, on lines 1 and 12. The one on line 1 is discarded, replaced by the one on line 23.

Post-Processor Commands

Each `processor` must provide a series of commands to execute on a given text file to extract pertinent information in a delimiter-separated-value format. A simple example might be to use `grep` to search for a line that contains a pair of numbers, and `awk` to format the line. The `logic` might look like this:

```
grep "x-y" ${CURRENT_FILE} | awk "{print $2, $3}"
```

> Note
>
> `${CURRENT_FILE}` is defined to be the text file against which a `processor` is executed

Assuming that the target file contains `x-y 1 5`, the output of this command would be `1 5`.

`delimiter` specifies the field delimiter to use when splitting the `logic` output into columns. In the above example, `delimiter` should have been a single space. `delimiter` may be a regular expression matching the syntax of a *highlighter* rule's pattern.

Expected Command Output

The output of a post-processor command is expected to be like an Excel spreadsheet: a two-dimensional matrix of cells, populated with the textual output of the command. When extracting the data to create plot series, the cells can be referenced using an Excel-like pattern: `[first column][first row]:[last column][last row]`, where columns are letters and rows are numbers:

```
A1: column 1, row 1

D17: column 4, row 17

C7:F10: column 3, row 7, through column 6, row 10
```

When selecting a whole row, the pattern does not need to specify the column (and vice versa):

```
E:E: column 5, all rows

3:3: all columns, row 3
```

One difference from an Excel spreadsheet is that the processor cell patterns have the ability to specify a wildcard, `?`. This allows the user to select a range of cells without having to know the start or end row and/or column:

A1:A?: column **1**, row **1**, through column **1**, last row (equivalent to A:A) C7:?**10**: column **6**, row **7**, through last column, row **10**

B?:E5: column **2**, first row, through column **5**, row **5** (equivalent to B1:E5)

?**2**:D6: first column, row **2**, through column **4**, row **6** (equivalent to A2:D6)

G8:??: column **7**, row **8**, through last column, last row (the end of the spreadsheet)

Plot Series

The end-goal of the post-processor capability is to automatically generate a plot of the data extracted by `logic`. These plots will either be bar, color map, line, or scatter plots. One `logic` can be used to create any number of series:

```
processor("Processor 1") { delimiter = " "

logic = 'grep "x-y" ${CURRENT_FILE} | awk "{print $2, $3}"'


bar("Bar series name") {...} colormap("Color map series name") {...} line("Line series name") {...}
scatter("Scatter series name") {...}

}
```

Bar Series

A bar series is a typical bar chart with categories along the x-axis and bars extending along the y-axis. The following is an example of a bar series:

```
bar("Bar series name") { key_labels = "a:a" keys = "b:b"

values = "c:c"

}
```

`key_labels` is a cell-selection pattern that specifies which cells should contain the categories' names. In this example, the categories' labels should be in the first column.

`keys` specifies which cells should contain the categories'' keys (how to order the bars on the chart). In this example, the categories' keys should be in the second column.

`values` specifies which cells should contain the categories' values. In this example, the categories' values should be in the third column.

Note The above example could be used with the following "spreadsheet" data to create a bar series:

```
Bar1 1 1
Bar2 2 3
Bar3 3 5
Bar4 4 7
Bar5 5 5
Bar6 6 3
Bar7 7 1
```

It would look something like this:

Color Map Series

A `colormap` series is a two-dimensional matrix of color-coded cells. Data can be read in rectangular or lower-triangular matrix forms. Cell colors are selected automatically based on the cells' values relative to the range of values found in the matrix. For example, if the color scale ranges from red (low) to yellow (high), the minimum value would map to red, the maximum value would map to yellow, and a value in the middle might map to orange.

The following is an example of a `colormap` series:

```
colormap("Color map series name") { data = "b2:??"

key_labels = "b1:?1"

type = "lower_triangular" value_labels = "a2:a?"

}
```

`data` specifies which cells should contain the matrix data. In this example, the data should start in row 2, column B (2), and extend to the bottom right corner.

`key_labels` specifies which cells should contain the labels for the columns. In this example, the labels should be in the first row.

`type` specifies whether the matrix should be read as a full (rectangular) matrix or a lower-triangular matrix. It is assumed to be a full matrix if this field is not present.

`value_labels` specifies which cells should contain the labels for the rows. In this example, the labels should be in the first column, starting with the second row.

> Note
>
> The above example could be used with the following "spreadsheet" data to create a color map series:

```
row1 1
row2 0.75 1
row3 0.5 0.25 1
row4 0.25 0.125 0.0625
     1
row5 0 -0.25 -0.5 -    1
     0.75
```

It will look something like this:
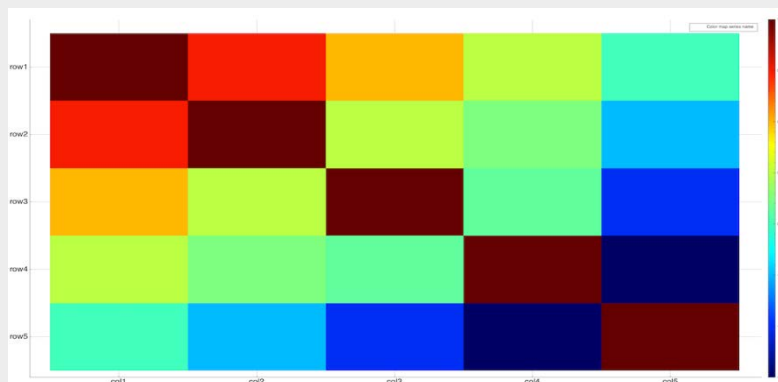
Line Series

A `line` series is a typical line plot with a collection of key-value (x-y) pairs with lines connecting them. The keys/values can also have associated uncertainty values; if provided, the number of keys/values/respective uncertainties should match. The following is an example of a `line` series:

```
line("Line series name") { keys = "a:a" keys_uncertainty = "b:b" values = "c:c" values_uncertainty = "d:d"
```

`keys` specifies which cells should contain the keys (x-coordinate) for the data points.

`keys_uncertainty` specifies which cells should contain the keys' uncertainties. This field is optional.

`keys_uncertainty_high` specifies which cells should contain the keys' upper-bound uncertainties. This field is optional.

`keys_uncertainty_low` specifies which cells should contain the keys" lower-bound uncertainties. This field is optional.

`values` specifies which cells should contain the values (y-coordinate) for the data points.

`values_uncertainty` specifies which cells should contain the values' uncertainties. This field is optional.

`values_uncertainty_high` specifies which cells should contain the values' upper-bound uncertainties. This field is optional.

`values_uncertainty_low` specifies which cells should contain the values' lower-bound uncertainties. This field is optional.

> **Note**
>
> Currently, there is no restriction on whether `keys_uncertainty` can be specified at the same time as `keys_uncertainty_low` and `keys_uncertainty_high` (similarly, with `values*`). `*low` and `*high` currently take precedence, but this is not guaranteed to always be the case.
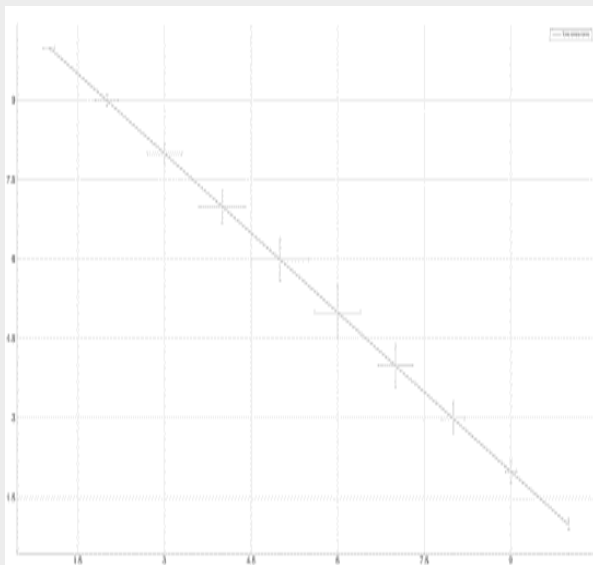
```
2 0.2 9 0.1
3 0.3 8 0.2
4 0.4 7 0.3
5 0.5 6 0.4
6 0.4 5 0.5
7 0.3 4 0.4
8 0.2 3 0.3
9 0.1 2 0.2
```

> **Note**
>
> The above example could be used with the following "spreadsheet" data to create a line series:
>
> ```
> 1 0.1 10 0
> 10 0 1 0.1
> ```
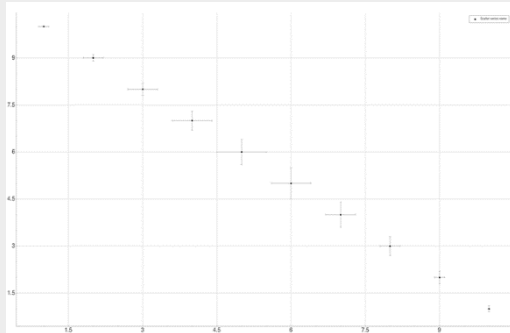>
> It will look something like this:

Scatter Series

A `scatter` series is like a line series, except the lines are hidden and the data points are visible.

Note

The above line series will look something like this if it were a scatter series:

Conditionally Enabling Post-Processors

By default, all post-processors are enabled and available for any text file the user may have open in Workbench. This may not always be desirable, particularly as the list of available post-processors grows. The following fields are provided to assist with paring down the list to a more manageable (and contextually relevant) set for a given file:

- `extensions`: a list of file extensions by which to coarsely filter the following post-processors
- `filter_pattern`: a case-sensitive, `grep`-supported regular expression by which to more finely filter the following post-processors

These fields can appear more than once in a post-processor file. Any post-processors following one of these fields will be filtered according to the `extensions` or `filter_pattern` specified until another is seen. The following will demonstrate this feature:

```
processor("Processor 1") {...}


extensions = [out] filter_pattern = "first pattern" processor("Processor 2") {...}


filter_pattern = "(seco|2)nd pattern" processor("Processor 3") {...}


extensions = [txt] processor("Processor 4") {...}


extensions = [] filter_pattern = ""
processor("Processor 5") {...}
```

- `Processor 1` will be enabled for all text files since there are no preceding filters
- `Processor 2` is only enabled for `*.out` files that contain `first pattern`
- Processor 3 is only enabled for *.out files that contain second pattern or 2nd pattern
- Processor 4 is only enabled for *.txt files that contain second pattern or 2nd pattern
- `Processor 5` will be enabled for all text files since `extensions` and `filter_pattern` are "reset" with an empty list and empty string, respectively

Organizing Post-Processors for Use in Workbench

By default, all post-processors are grouped together in a single list where the user may select one to be run against a given text file. As the number of available post-processors grows, this list will become unwieldy, so it would be convenient to be able to organize post-processors according to context/relevance. The `hierarchies` field does just this.

When the user is viewing a text file, there is a menu containing the list of available processors (it is disabled if there are none). `hierarchies` allows the user to specify a list of "paths" under which to place the following `processors` in the menu. The following will demonstrate this feature:

```
processor("Processor 1") {...}
processor("Processor 2") {...}


hierarchies = ["Group 1"] processor("Processor 3") {...}


hierarchies = ["Group 1/Subgroup 1", "Group 2"] processor("Processor 4") {...}


hierarchies = ["Group 1/Subgroup 1", "Group 2/Subgroup 1"] processor("Processor 5") {...}
```

The following is an approximate visual representation of the resultant menu structure:

```
Processors
Processor 1
Processor 2
Group 1
Processor 3
Subgroup 1
Processor 4
Processor 5
Group 2
```

`Group 1` and `Group 2` are submenus of the main `Processors` menu, and each has its own submenu, Subgroup 1.

Configurable Views

Workbench allows the user to configure open views in a way only restricted by the amount of screen estate available. This allows a Workbench to be configured to a layout most suitable to the user.
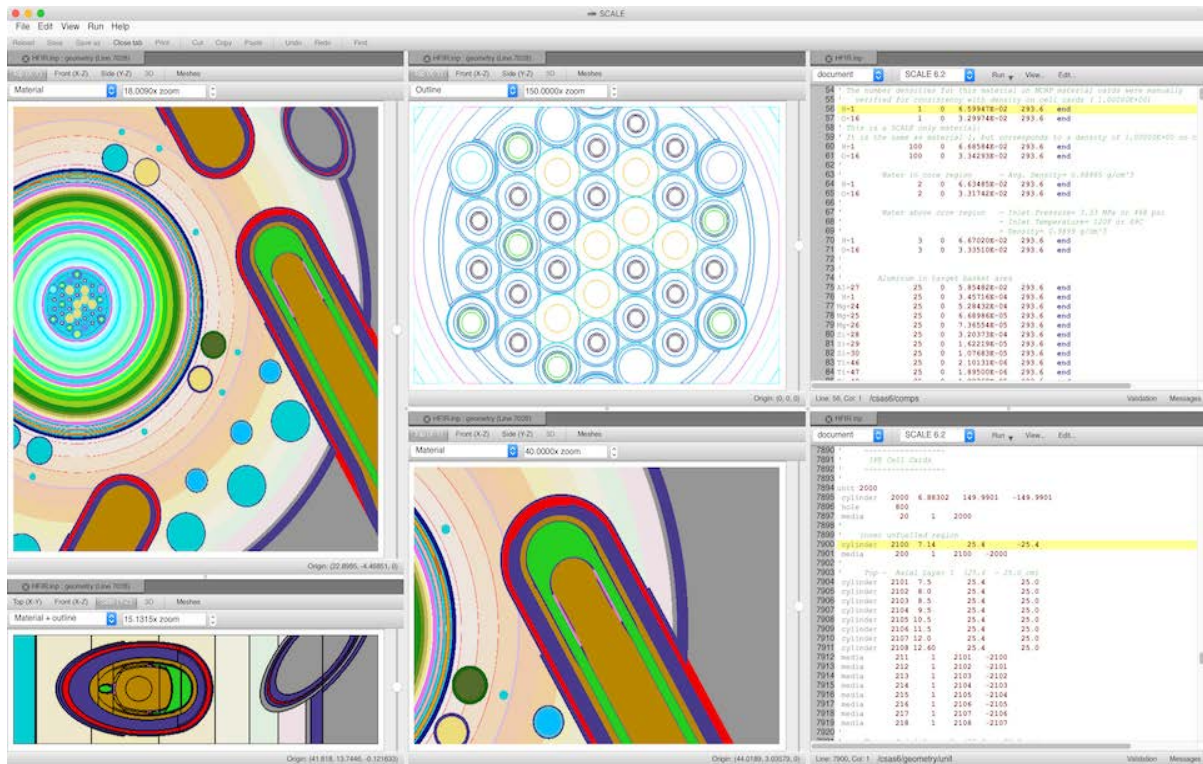
Split Views

This section discusses splitting views to the top, bottom, left, or right, and resizing splits.

It is often desirable to view multiple files at the same time. This is easily done by dragging and dropping a tab to the top, bottom, left, or right of the area to be split.

Steps:

1. Drag by left-clicking and holding the tab while moving the mouse.
2. Drag the tab to the top, bottom, left, or right, and drop by releasing the left mouse button.

Note

This panel in which the drag is occurring will provide some visual feedback by highlighting the region where the split will occur.

Split Top

Left-click the tab in the tab group where a new split is desired, and drag to the middle of the top border of the section of the tab where the split is desired.
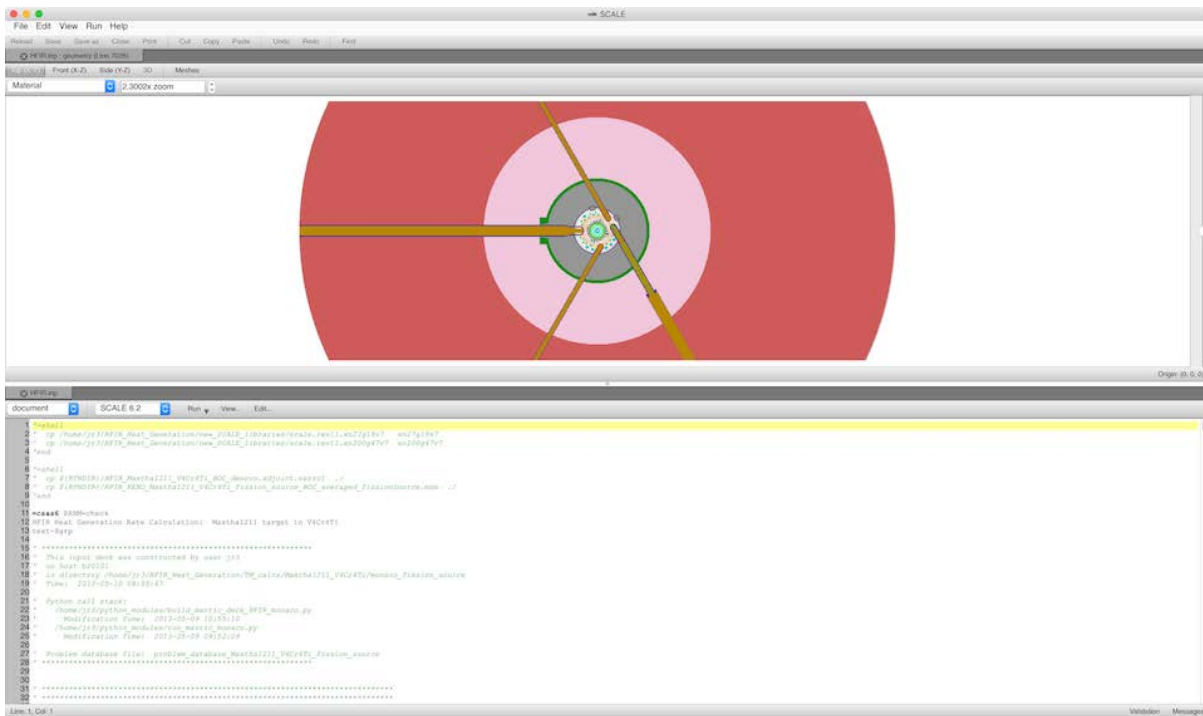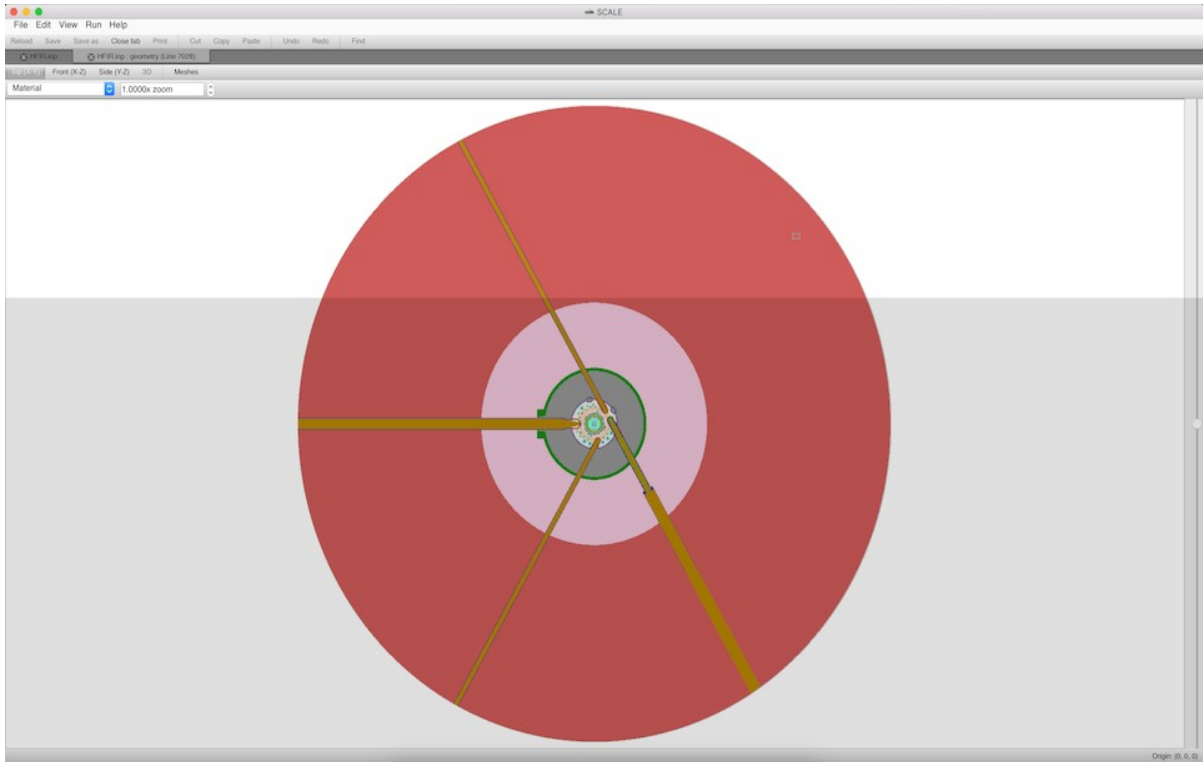
Split Bottom

Left-click the tab from the tab group where a new split is desired, and drag to the middle of the bottom border of the section of the tab where the split is desired.

Split Left

Left-click the tab from the tab group where a new split is desired, and drag to the middle of the left border of the section of the tab where the split is desired.

Split Right

Left-click the tab from the tab group where a new split is desired, and drag to the middle of the right
border of the section of the tab where the split is desired.

Split Resizing

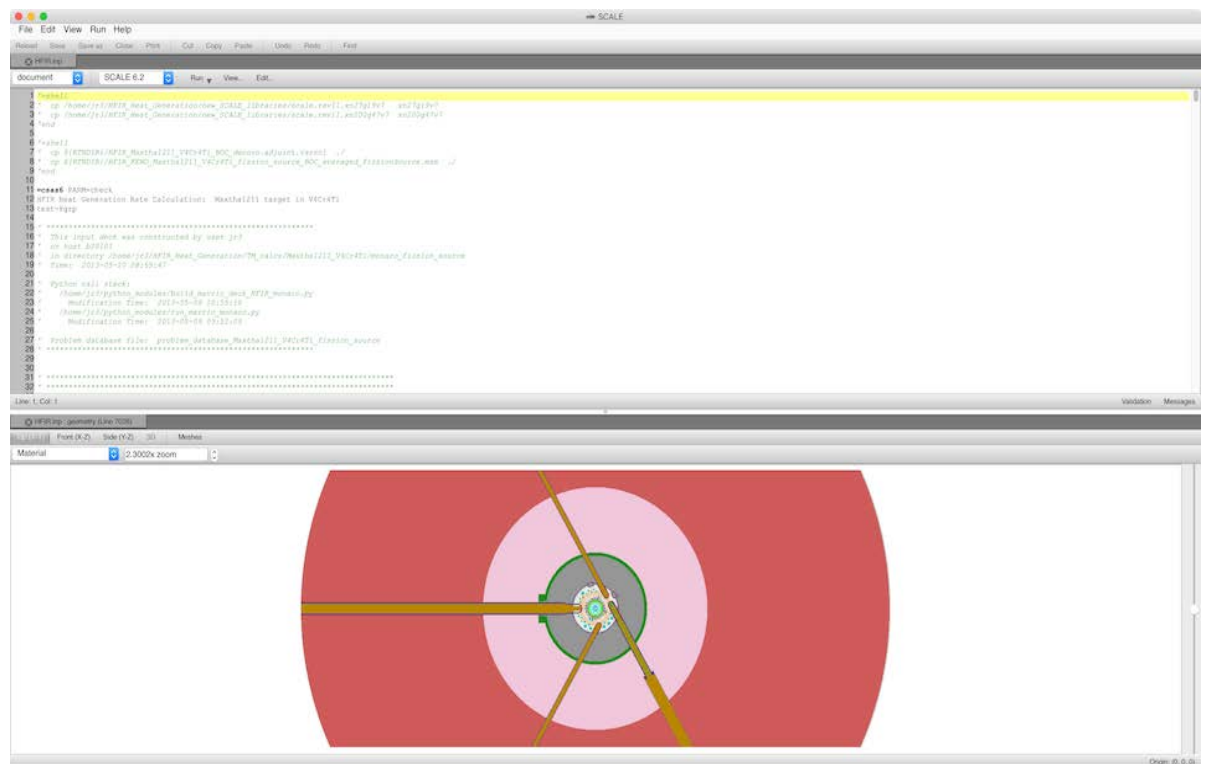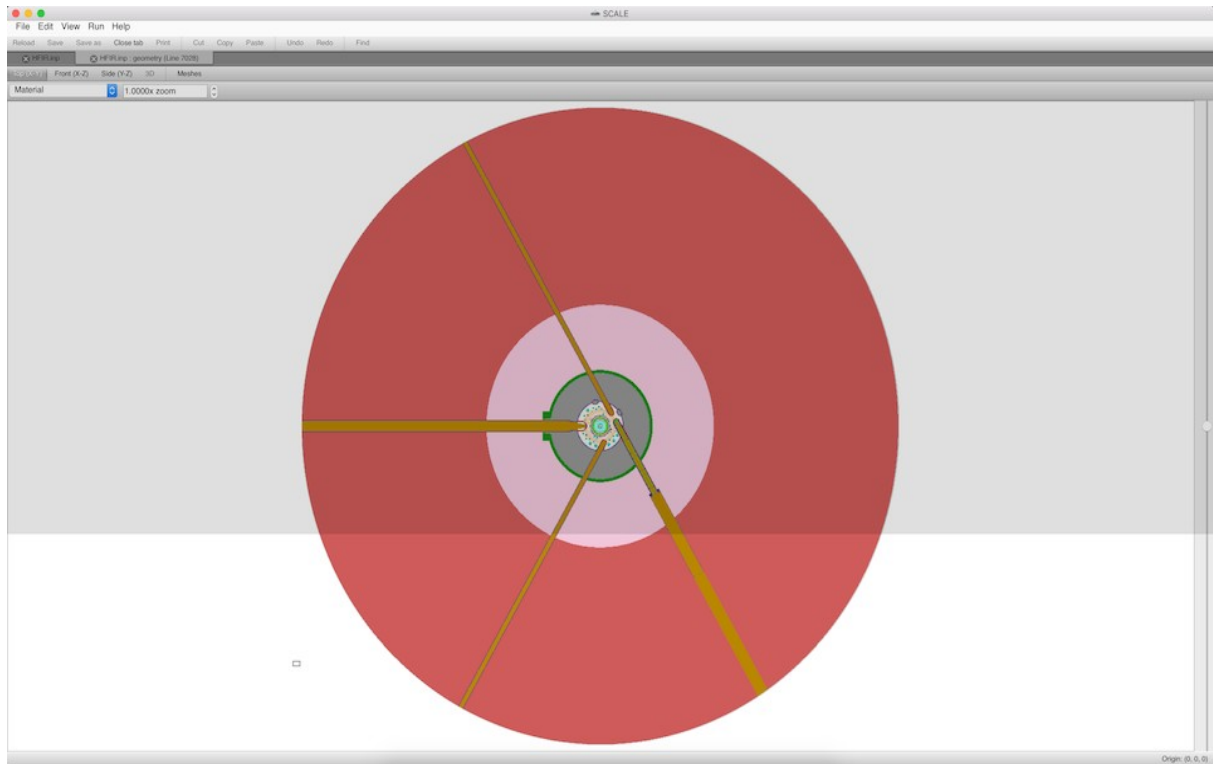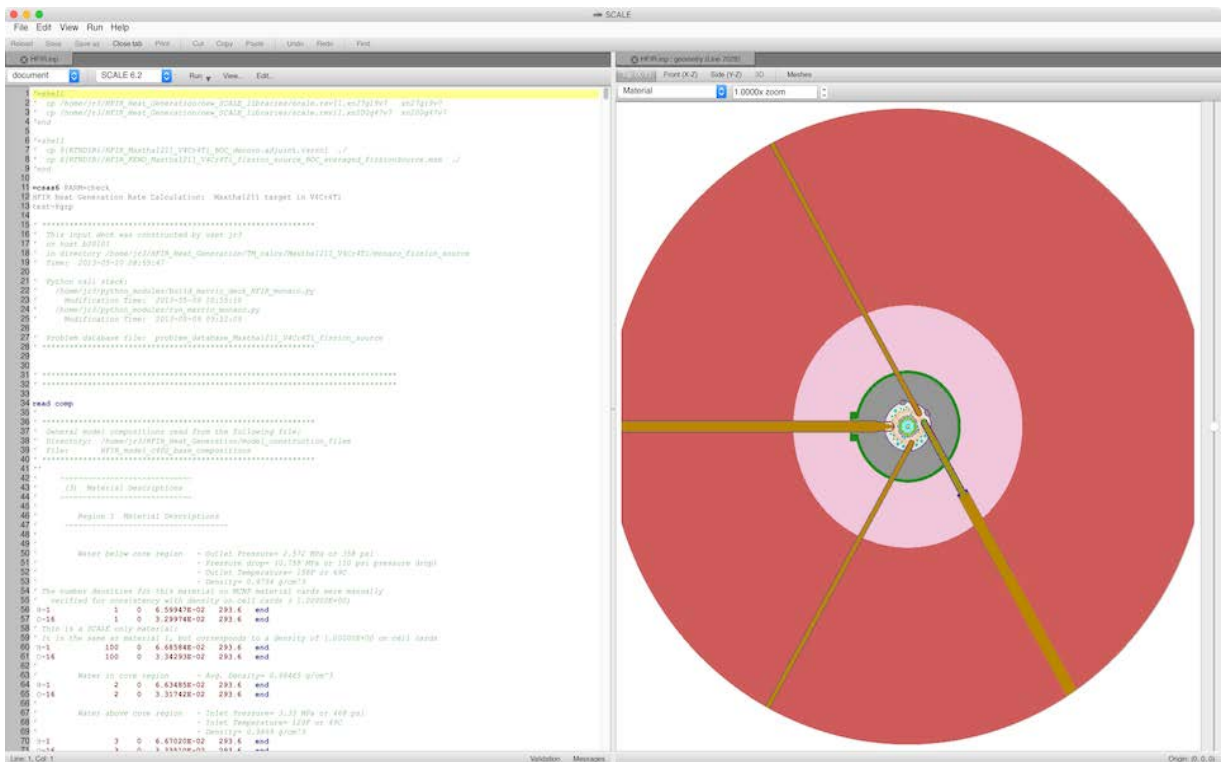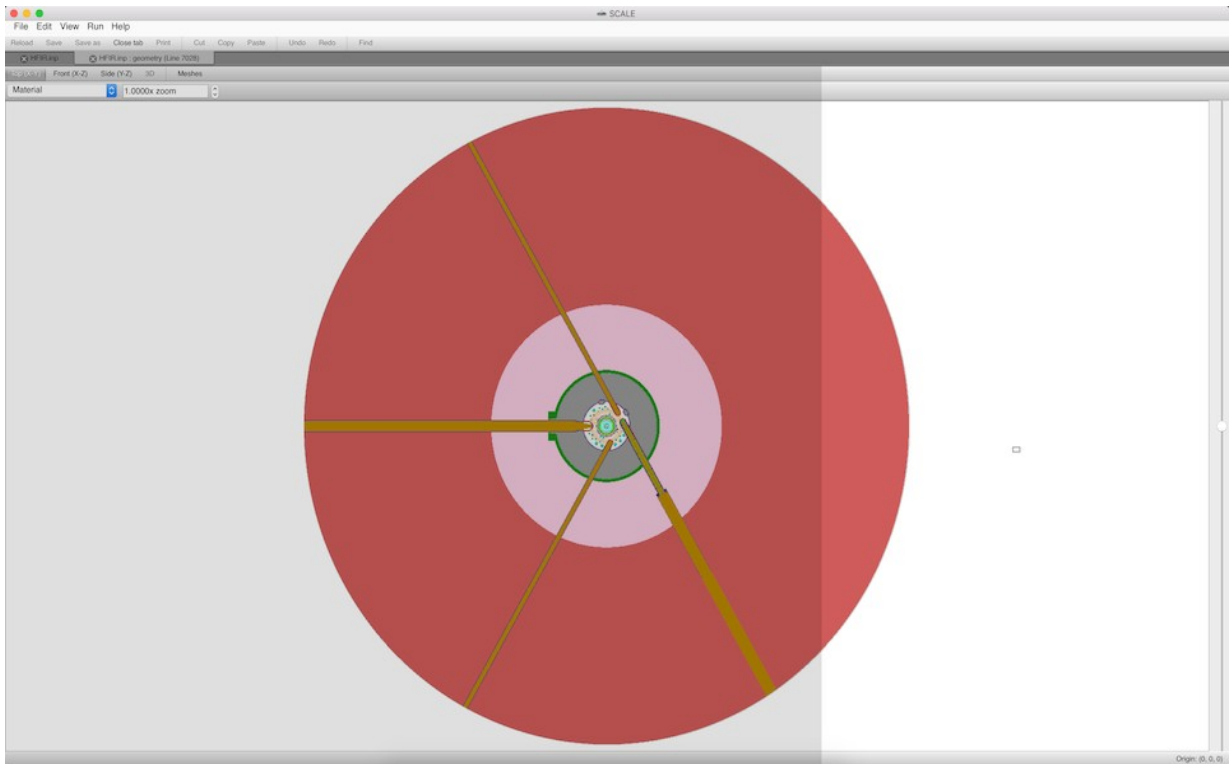Resizing splits can often be desired and is easily done. Between each split resides a split widget; left-click this widget and drag it in the direction of the desired resize.

Data Plotting

Workbench provides data plotting capabilities for many of the plot formats produced by SCALE. The plot interface is intended to be fast, unobtrusive, and intuitive, but it is still under development.

2D Plot Interface Controls

The 2D Plot interface allows zooming, panning, and other interactions expected from a 2D plotting package, as detailed below. Most data produced by SCALE is 2D graph data, as seen above; covariance and correlation matrices can also be visualized via this plot interface, as seen below.

Zooming

Zooming is performed via the mouse scroll wheel. Scrolling down, zooms in, and scrolling up, zooms out. Additionally, right clicking on the plot will present a context menu which includes the item "Fit graph," which will zoom the plot to the extents of the data. Upon mousing over the data, an overlay item displays the data coordinate under the mouse. As illustrated above, at the point under the mouse, the generation run and k_{eff} are 79.5105 and 0.828436, respectively. Clicking a graph on a plot will highlight the graph data and update the hover item with the graph's name.

Panning

Panning is performed by conducting a mouse click and drag. During the mouse drag, the plot will automatically update. Additionally, right clicking on the plot will present a context menu that includes the item "Fit graph" which will zoom the plot to the extents of the data.

Saving

The plot interface allows saving the current plot view in the portable document format (PDF), portable network graphics (PNG) format, the joint photographic expert group (JPG) image format, or the SCALE plot [interactive] format (SPF). SPF is only available for 2D graph plots, but it provides a mechanism for persisting configured plot views and property settings.

Plot Properties

Plot properties are accessible via right clicking the plot and selecting "Plot options."

Chart

This facilitates toggling the plot wide axis visibility and plot title.

Axis

This function facilitates configuring the plot axis label, scale, range, and grid lines.

Graph

Graph facilitates configuring a specific graph's name, line style, color, weight, scatter style, pen, and error bar visibility.

Legend

Legend facilitates changing legend visibility, font family, size, and look (bold, italic, underlined, etc.).

Plot Tips

This is a list of features that are not immediately visible, but typically hidden in context menus or key combinations.

Plottable Navigation Items

When a file is opened, any plottable item is represented in the navigation panel with a navigation item that includes a plot icon. Double clicking this navigation item will plot the data represented by the navigation item.

Adding a Graph to an Existing Plot

You can add data as a graph to an existing plot by finding the data item in the navigation panel and right clicking the item to acquire the its context menu. From this context menu, you can select "New Plot," or if any other plots with matching axis names are present, "Add graph to 'some other plot,'" where "some other plot" is the existing plot's title.

Copying the Plot's Data Table To copy the presented plot data to Excel, copy all cells. To select the cells, right click the table and left click "Select All," and subsequently, "Copy" or clicking the upper left most cell and right clicking the selection and selecting "Copy." After copying the table, paste into Excel as usual.

Supported Plot Formats

AMPX Continuous Energy Cross Sections

SCALE's Continuous Energy (SCE) cross section data can be plotted by selecting File -> Open SCE Library.... Navigate to the SCALE data directory and select ce_v*_endf.xml files for plotting (e.g., ce_v7.1_endf.xml). Upon selection, a few moments may be needed to load the header information, after which the navigation panel will list the nuclides available for inspection. Double left clicking the desired nuclide, temperature, and reaction will display a plot of the cross section over energy.

AMPX Multigroup Cross Sections

SCALE's Multigroup cross section data can be plotted by selecting File -> Open multigroup library.... Navigate to the SCALE data directory and select scale.rev*.xn* files for plotting (e.g., scale.rev04.xn252v7.1). Upon selection, a few moments may be needed to load the information after which the navigation panel will list the nuclides available for inspection. Double left clicking the desired nuclide, radiation (neutron or gamma), reaction, or transfer array with temperature and reaction, will display the plot.

ORIGEN (F71) Concentration Plotting

ORIGEN concentration files (F71) can be opened when the file has the *.f71 extension. Select File -> Open... and select any file with the *.f71 extension.

ORIGEN Opus (PLT) Concentration and Spectra Plotting

Opus plot files (PLT) can be opened when the file has the *.plt extension. Select File -> Open... and select any file with the *.plt extension.

ORIGEN Gamma Line Plotting

SCALE's gamma data can be plotted by selecting File -> Open ORIGEN gamma data.... Navigate to the SCALE data directory and select origen.rev*.*gam.data file for plotting (e.g., origen.rev04.mpdkxgam.data) typically inside the origen_data directory within the SCALE data directory. Once the data are loaded, double left clicking the nuclide data will create a new plot.

Ptolemy Plot (PTP) General 2D Plotting

Ptolemy plot files (PTP) can be opened when the file has the *.ptp extension. Select File -> Open... and select any file with the *.ptp extension. Double clicking the document navigation item will create a plot with all available graphs plotted. Alternatively, double left clicking a single graph item create a new plot of just that data.

SCALE Plot Format (SPF) General 2D Plotting

SCALE Plot Format (SPF) can be opened when the file has the *.spf extension. Select File -> Open... and select any file with the *.spf extension. Double clicking the document navigation item will create a plot with all available graphs plotted. Alternatively, double left clicking a single graph item create a new plot of just that data.

Covariance (COVERX) Matrix Plotting

SCALE's covariance data can be plotted by selecting File -> Open covariance library.... Navigate to the SCALE data directory and select scale.rev*.*groupcov files for plotting (e.g., scale.rev08.56groupcov7.1). Once the data are loaded, double left click a nuclide-reaction to nuclide-

reaction item from the navigation panel, which will display the correlation coefficient matrix. Alternatively, right clicking on the desired navigation item will display a context menu where you can select correlation coefficient matrix, covariance matrix, New plot - std dev by group, or New plot - std dev by energy.

Sensitivity Data

Sensitivity Data Files (sdf) can be opened when the file has the *.sdf extension. Select File -> Open... and select any file with the *.sdf extension.

Using VisIt

As a result of collaboration with Berkeley Lab, the VisIt data visualization tool has been embedded into Workbench. This provides a robust set of 2D and 3D data visualization capabilities not previously available. If Workbench is built with VisIt support, then the standard VisIt interface will be embedded in (docked to) the right side of Workbench's main window. It can be docked to all four sides or popped out as a free-floating window.

Note

VisIt works with a fixed maximum number of rendering windows, currently set to 16 in Workbench. To ensure at least one rendering window exists, one window will be reserved and not used, leaving only 15 tabs available at a time.

To start a visualization workflow with VisIt, select `File : New Visualization` to create a new tab that is dedicated to displaying VisIt-supported content. Once the tab is created, VisIt is able to use the tab to visualize data. At this point, all VisIt-related functionality is provided by VisIt's own interface. For more information on how to use VisIt, please see their documentation at the VisIt website (http://visit.llnl.gov).